Desbravando SOLID

Práticas avançadas para códigos de qualidade em Java moderno

Casa do Código | alura

ALEXANDRE AQUILES

Sumário

- ISBN
- Prefácio por João Júnior
- Agradecimentos
- Sobre o livro
- 1. Para que serve Orientação a Objetos?
- 2. Conhecendo o Cotuba
- 3. Princípio da Responsabilidade Única (SRP): classes coesas
- 4. Princípio da Inversão de Dependências (DIP): dependências estáveis
- 5. Princípio Aberto/Fechado (OCP): objetos flexíveis
- 6. OCP potencializado: plugins
- 7. Princípio de Substituição de Liskov (LSP): herança do jeito certo
- 8. Princípio da Segregação de Interfaces (ISP): clientes separados, interfaces separadas
- 9. Um pouco de imutabilidade e encapsulamento
- 10. Princípios de Coesão e Acoplamento de Módulos
- 11. Além da linha de comando: o Cotuba Web
- 12. Módulos com o Java Platform Module System (JPMS)
- 13. Arquitetura Hexagonal: uma arquitetura centrada no domínio
- 14. Apêndice: O código inicial do Cotuba
- 15. Apêndice: Referências

ISBN

Impresso: 978-85-5519-309-5

Digital: 978-85-5519-310-1

Caso você deseje submeter alguma errata ou sugestão, acesse http://erratas.casadocodigo.com.br.

Prefácio por João Júnior

Este livro é único, essencial e prático!

Toda leitura é um diálogo entre o leitor e o autor do livro. Quando o livro é sobre um assunto técnico, além do diálogo, eu acredito que devemos fazer um debate com o autor. Para quem tem o privilégio de conhecer o Alexandre, a leitura deste livro é exatamente como um debate com ele. Não apenas respostas simplórias são dadas, mas o contexto de cada decisão, além dos contrapontos, quando necessário, são apresentados pelo autor.

Logo após aprendermos a programar, nos primeiros anos de nossa carreira, emerge a necessidade de organizar o nosso código para ser fácil de entender, ler e manter. Este livro visa ensinar formas e conceitos de como fazer isso, através de Orientação a Objetos e SOLID.

Se você é um desenvolvedor ou uma desenvolvedora sem muita experiência, você verá os fundamentos de Orientação a Objetos: encapsulamento, polimorfismo, herança e composição. Além disso, você será exposto(a) a bons e maus exemplos de uso desses conceitos. Isso será um grande diferencial na sua carreira, e é muito difícil encontrar materiais sobre esses conceitos tão bem descritos e aplicados em exemplos reais.

Por outro lado, se você é experiente, você vai aprender como usar Orientação a Objetos e diversas técnicas e conceitos para simplificar o código do seu projeto. Você vai não apenas aprender conceitos de forma isolada, mas também será guiado(a) a como tomar decisões sobre como e quando usar esses conceitos. Você também vai aprender sobre quando não deve usar muitos dos conceitos e técnicas apresentados aqui.

Embora os exemplos apresentados sejam na linguagem Java, este livro é essencial para desenvolvedores de todas as linguagens. Se você programa em Java, poderá aplicar os exemplos diretamente

em seus códigos. E se você utiliza outra linguagem de programação, não será muito difícil adaptar o que aprendeu aqui.

Além de todo o conhecimento apresentado, um grande diferencial é que ele usa todos os conceitos e técnicas apresentados em uma aplicação real. Então, você vai começar com um código que já está em produção e, à medida que passar pelos capítulos, vai refatorar o código para torná-lo mais legível e extensível. Essa é uma das características que torna este livro único, essencial e prático. O autor também nos presenteia com inúmeras referências para continuarmos nossos estudos.

Então, tenha um excelente debate!

João Júnior, Tech Lead na Shopify

Agradecimentos

Agradeço à D. Clores e ao Sr. Sipriano, meus pais, por serem tanto exemplo de vida como raiz que nutre.

Agradeço à Caelum e, em especial, a Alberto Souza que incentivou e investiu na criação do curso que originou este livro.

Agradeço aos alunos e alunas que fizeram o curso e, por meio de seus feedbacks, ajudaram em diversas melhorias.

Agradeço à equipe da Casa do Código, pelo trabalho de revisão e adaptação desse material em um livro.

Sobre o autor

Alexandre Aquiles é Engenheiro de Software desde 2005, tendo desenvolvido software para indústrias como Logística, Governo, Mineração, Seguros e Educação. Acredita que softwares devem ser desenvolvidos de forma iterativa e incremental, entregando continuamente código de valor e com qualidade. Especializado na plataforma Java, tem mais de 6500 horas como instrutor de cursos sobre Orientação a Objetos, Arquitetura de Software, Testes Automatizados e TDD, Integração de Sistemas e Microsserviços. É coautor do livro *Controlando versões com Git e GitHub* publicado em 2014 pela Casa do Código.

Sobre o livro

Este livro é a adaptação do curso *Práticas de Design e Arquitetura de Código* da Caelum, criado pelo autor em 2018 e ministrado para centenas de alunos e alunas desde então.

No decorrer do livro, vamos melhorar o código do Cotuba, uma aplicação que gera e-books nos formatos PDF e EPUB a partir de arquivos Markdown, e também adicionaremos algumas funcionalidades à aplicação.

À medida que formos refatorando e adicionando funcionalidades ao Cotuba, estudaremos assuntos avançados de Orientação a Objetos.

O capítulo 1 apresenta uma contextualização dos princípios SOLID em relação a outras técnicas.

No capítulo 2, conheceremos o código inicial do Cotuba, que será refatorado nos capítulos posteriores.

No capítulo 3, estudaremos o *Single Responsibility Principle* (Princípio da Responsabilidade Única), aplicando-o ao Cotuba.

No capítulo 4, abordaremos o *Dependency Inversion Principle* (Princípio da Inversão das Dependências) e o design pattern Factory.

No capítulo 5, é a vez de estudarmos o *Open/Closed Principle* (Princípio do Aberto/Fechado) e os design patterns Command e Strategy.

No capítulo 6, criaremos plugins com a Service Loader API, estudando um caso extremo do *Open/Closed Principle*.

No capítulo 7, corrigiremos violações do *Liskov Substitution Principle* (Princípio de Substituição de Liskov) no Cotuba.

No capítulo 8, vamos explorar o *Interface Segregation Principle* (Princípio da Segregação de Interfaces).

O capítulo 9 trata de assuntos importantes em Orientação a Objetos como imutabilidade e encapsulamento, além de implementar os design patterns Builder e Iterator.

No capítulo 10, discutiremos sobre modularização, abordando os princípios de Coesão e Acoplamento de módulos e implementando módulos Maven.

No capítulo 11, verificaremos a flexibilidade do design de código do Cotuba, utilizando-o em uma aplicação Web e ajustando alguns detalhes.

No capítulo 12, veremos como usar o Java Platform Module System (JPMS), disponível a partir do Java 9, para reforçar o encapsulamento dos módulos do Cotuba.

No capítulo 13, definiremos o que é uma Arquitetura Hexagonal e como o código refatorado do Cotuba se encaixa nesse estilo arquitetural.

Ao final de cada capítulo, haverá um QR Code de um vídeo que demonstra o passo a passo de código. Além disso, teremos um apêndice com o código inicial do Cotuba e outro com as referências bibliográficas.

Para quem é o livro

Este livro é focado em desenvolvedores e desenvolvedoras familiarizados com a linguagem Java e com os fundamentos da Orientação de Objetos, como classes, objetos, atributos, métodos, herança, polimorfismo, interfaces e que desejam estudar técnicas avançadas de design de código.

Neste livro, vamos utilizar recursos do Java 17. É recomendada a utilização de uma IDE, como IntelliJ ou Eclipse, para seguir os exemplos do livro. Além disso, utilizaremos a ferramenta Maven, na versão 3.8 ou posterior.

Nos exemplos de linha de comando, utilizaremos comandos disponíveis em um Unix como qualquer distribuição Linux ou MacOS. Para executá-los em uma máquina Windows, utilize ferramentas como WSL, Cygwin ou Git Bash. Além disso, não deixe de instalar programas leitores de PDFs e EPUBs!

CAPÍTULO 1

Para que serve Orientação a Objetos?

1.1 Modelagem e dependências

Você aprendeu Orientação a Objetos (OO). Entendeu classes, objetos, atributos, métodos, herança, polimorfismo, interfaces. Aprendeu algumas soluções comuns para problemas recorrentes estudando *Design Patterns*. Mas como e quando usar OO?

Sem dúvida, a resposta tem a ver com organizar o código, minimizando o impacto de mudanças no médio/longo prazo.

No artigo *The Principles of OOD* (MARTIN, 2005), Robert Cecil Martin, vulgo Uncle Bob, descreve duas abordagens complementares no uso de OO:

- Criar um **modelo de domínio** no seu código.
- Gerenciar as **dependências** do seu código.

> Uncle Bob é notável por suas influentes visões técnicas, que vamos discutir e aplicar com profundidade. O mesmo Uncle Bob é famigerado por suas funestas opiniões políticas em redes sociais, que não serão abordadas e que são completamente repudiadas por este que vos escreve.

OO como ferramenta de modelagem do domínio

OO é uma ótima ferramenta para representar, em código, os conceitos do problema que estamos resolvendo. É fundamental para o sucesso de um projeto de software selecionar entidades de negócio e criar um *domain model*, ou modelo de domínio, que as "traduza" para uma linguagem de programação.

Um bom *domain model* é o foco de metodologias e técnicas como:

- *Class-Responsibility-Collaboration* (CRC) - proposta por Ward Cunningham e Kent Beck para explorar os possíveis objetos de

um sistema e suas responsabilidades.
- *Feature-Driven Development* (FDD) - uma metodologia ágil criada por Peter Coad e Jeff De Luca que usa funcionalidades de um sistema como guia para a modelagem dos objetos.
- *Domain-Driven Design (DDD)* - descrita por Eric Evans em que a modelagem é focada em representar, em código, a linguagem dos especialistas de negócio.

OO como ferramenta de gerenciamento de dependências

OO também é uma ótima maneira de evitar código "amarrado" demais, controlando as dependências e minimizando o acoplamento. Um bom código OO, com as dependências administradas com cuidado, leva a mais flexibilidade, robustez e possibilidade de reúso.

Dependências bem gerenciadas, independentes de domínio, são o foco de técnicas como:

- *GRASP Patterns/Principles* - um guia criado por Craig Larman com técnicas como baixo acoplamento, alta coesão, indireção, polimorfismo, entre outras.
- *Dependency Injection* - uma técnica em que um objeto recebe os outros objetos dos quais depende por meio de um framework como Spring, Weld ou Dagger.
- *Design Patterns* - um catálogo de soluções comuns para problemas recorrentes descrito por Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides. É parte de um movimento mais amplo que inclui *patterns* arquiteturais, de integração de sistemas, entre diversos outros. Abordaremos alguns *design patterns* ao decorrer deste livro.
- *Princípios SOLID* - definidos por Uncle Bob e que estudaremos com profundidade neste livro.

"Esses princípios (SOLID) expõem os aspectos de gerenciamento de dependências do Design Orientado a Objetos (OOD), em oposição aos aspectos de conceitualização e modelagem. Isso não quer dizer que OO é uma ferramenta ruim para conceitualização do domínio do problema ou que não é um bom veículo para criar modelos. Certamente, muitas pessoas extraem valor desses aspectos de OO. Os princípios, porém, focam bastante no gerenciamento de dependências."

Uncle Bob, no artigo *The Principles of OOD* (MARTIN, 2005)

1.2 Os princípios SOLID

Nosso foco neste livro é aprofundar no estudo dos princípios SOLID de Orientação a Objetos, um acrônimo cunhado por Uncle Bob. Os cinco princípios SOLID são:

- ***S**ingle Responsibility Principle (SRP)*, o Princípio da Responsabilidade Única: uma classe deve ter um, e apenas um, motivo para ser modificada.
- ***O**pen/Closed Principle (OCP)*, o Princípio Aberto/Fechado: deve ser possível estender o comportamento de uma classe sem modificá-la.
- ***L**iskov Substitution Principle (LSP)*, o Princípio de Substituição de Liskov: subtipos devem ser substituíveis por seus tipos base.
- ***I**nterface Segregation Principle (ISP)*, o Princípio de Segregação de Interfaces: clientes não devem ser obrigados a depender de métodos que eles não usam.
- ***D**ependency Inversion Principle (DIP)*, o Princípio de Inversão de Dependências: módulos de alto nível não devem depender de módulos de baixo nível. Ambos devem depender de abstrações. Abstrações não devem depender de detalhes. Detalhes devem depender de abstrações.

No artigo *Getting a SOLID start* (MARTIN, 2009), Uncle Bob indica que os princípios SOLID não são *checklists*, nem leis ou regras. São bons conselhos vindos do senso comum de gente experiente, coletados em projetos reais ao longo do tempo. Não significa que sempre funcionam ou que sempre devem ser seguidos.

1.3 Os princípios de coesão e acoplamento de módulos

Uncle Bob foi além dos cinco princípios SOLID e descreveu mais alguns princípios relacionados a módulos que podem ser divididos em *Princípios de coesão* e *Princípios de acoplamento*.

Os princípios de coesão de módulos

Os três primeiros princípios de módulos descrevem como devemos organizar o conteúdo de um módulo. Ou seja, são princípios sobre **coesão**:

- *Release Reuse Equivalency Principle (REP)*, o Princípio da Equivalência entre Reúso e Release: a granularidade de reúso é a granularidade de entrega.
- *Common Closure Principle (CCP)*, o Princípio do Fechamento Comum: classes que são modificadas juntas devem estar no mesmo módulo.
- *Common Reuse Principle (CRP)*, o Princípio do Reúso Comum: classes que são usadas juntas devem estar no mesmo módulo.

Os princípios de acoplamento de módulos

Há mais três princípios de módulos, focados no **acoplamento** e em métricas para avaliar a estrutura de módulos de um sistema:

- *Acyclic Dependencies Principle (ADP)*, ou Princípio das Dependências Acíclicas: o grafo de dependências dos módulos não deve ter ciclos.
- *Stable Dependencies Principle (SDP)*, ou Princípio das Dependências Estáveis: dependências devem estar organizadas na direção da estabilidade.

- *Stable Abstractions Principle (SAP)*, ou Princípio das Abstrações Estáveis: abstração traz estabilidade.

Esses princípios de coesão e acoplamento de módulos serão detalhados em um capítulo posterior.

Para saber mais: a história dos princípios SOLID

Os dez (ou onze) mandamentos de OO

Robert Cecil Martin, o famoso Uncle Bob, listou os seus dez (na verdade, onze) mandamentos de OO, no grupo Usenet *comp.object* (MARTIN, 1995):

1. Entidades de software (classes, módulos etc.) devem ser abertas para extensão, mas fechadas para modificação *(o Open/Closed Principle - Bertrand Meyer).*
2. Classes derivadas devem ser usáveis através da interface da classe base sem a necessidade de o usuário saber a diferença *(o Liskov Substitution Principle).*
3. Detalhes devem depender de abstrações. Abstrações não devem depender de detalhes *(o Dependency Inversion Principle).*
4. A granularidade do reúso é a mesma que a granularidade da entrega. Apenas componentes que são entregues através de um *tracking system* podem ser efetivamente reusados.
5. As classes de um componente entregue devem compartilhar um fechamento (ou *closure*) comum. Isto é, se uma classe precisar ser modificada, todas as outras provavelmente precisarão ser modificadas. O que afeta um afeta todos.
6. As classes de um componente entregue devem ser reusadas juntas. Isto é, é impossível separar os componentes uns dos outros para que sejam menos que o todo.
7. A estrutura de dependências dos componentes entregues deve ser um grafo acíclico direcionado (DAG). Não podem existir ciclos.

8. Dependências entre componentes entregues devem ser na direção da estabilidade. O componente que recebe a dependência deve ser mais estável que o que usa a dependência.
9. Quanto mais estável um componente entregue é, mais deve consistir de classes abstratas. Um componente completamente estável deve consistir de nada mais que classes abstratas.
10. Onde possível, use patterns comprovados para resolver problemas de design.
11. Ao atravessar dois paradigmas diferentes, construa uma camada de interfaces que os separe. Não polua um lado com um paradigma do outro.

Os princípios tomam forma

Em 1996, Uncle Bob fez uma série de artigos na revista *C++ Report* sobre o que chamou de **princípios**:

- *Open-Closed Principle* (MARTIN, 1996a).
- *Liskov Substitution Principle* (MARTIN, 1996b).
- *Dependency Inversion Principle* (MARTIN, 1996c).
- *Interface Segregation Principle* (MARTIN, 1996d).
- *Granularity* (MARTIN, 1996e).
- *Stability* (MARTIN, 1997).

Em 2002, no livro *Agile Software Development: Principles, Patterns, and Practices* (MARTIN, 2002), ele adiciona o *Single Responsibility Principle*. Em 2004, em colaboração com Michael Feathers, ele reordena os princípios e cunha o acrônimo *S.O.L.I.D.* Em 2006, foi lançada uma versão em C# do livro original, com os mesmos princípios: *Agile Principles, Patterns, and Practices in C#* (MARTIN, 2006).

Agora que contextualizamos os princípios SOLID historicamente e em relação a outras técnicas avançadas de Orientação a Objetos, no próximo capítulo, vamos conhecer o Cotuba, a aplicação de

geração de e-books cujo código vamos melhorar utilizando esses princípios como guia!

CAPÍTULO 2

Conhecendo o Cotuba

O Cotuba é uma aplicação de linha de comando que converte arquivos escritos no formato Markdown (`.md`) em e-books nos formatos EPUB (`.epub`) e PDF (`.pdf`).

Neste capítulo, veremos como utilizar o Cotuba para transformar os arquivos `.md` de um diretório em um EPUB e/ou PDF. Além disso, estudaremos como o Cotuba foi implementado, identificando pontos de melhoria. Nos próximos capítulos, utilizaremos os princípios SOLID como guia para o aperfeiçoamento do código.

O "entregável" do Cotuba, a ser utilizado pelos usuários, é composto por um script `cotuba.sh` e um diretório `libs` , com um JAR do próprio Cotuba e diversos outros JARs de bibliotecas.

Para usar o Cotuba, devemos invocar o script `cotuba.sh` passando o formato de e-book desejado e um diretório que contenha arquivos `.md` . Algo semelhante a:

```
./cotuba.sh -d diretorio
```

Cada arquivo `.md` do diretório do livro, em ordem alfabética, será transformado em um capítulo do e-book e é possível configurar alguns detalhes de como o Cotuba gerará o e-book por meio das seguintes opções:

```
 -d,--dir <arg>      Diretório que contém os arquivos md. Default:
diretório atual.
 -f,--format <arg>   Formato de saída do e-book. Pode ser: pdf ou epub.
Default: pdf
 -o,--output <arg>   Arquivo de saída do e-book. Default: book.{formato}.
 -v,--verbose        Habilita modo verboso.
```

2.1 Usando o Cotuba

Considere que, no diretório `livro-exemplo`, temos o arquivo `01-pra-que-serve-oo.md` com o seguinte conteúdo:

```
# Para que serve OO?

## Modelagem e dependências

Você aprendeu **Orientação a Objetos**.

Entendeu classes, objetos, atributos, métodos, herança, polimorfismo,
interfaces.

Aprendeu algumas soluções comuns para problemas recorrentes estudando
alguns _Design Patterns_.
```

Para gerar um PDF a partir do diretório `livro-exemplo`, devemos executar:

```
./cotuba.sh -d livro-exemplo
```

Teremos como saída:

```
Arquivo gerado com sucesso: book.pdf
```

Um arquivo `book.pdf` será gerado e conterá um capítulo com título, subtítulo, os parágrafos e marcações em negrito e itálico.

Para que serve OO?

Modelagem e Dependências

Você aprendeu Orientação a Objetos.

Entendeu classes, objetos, atributos, métodos, herança, polimorfismo, interfaces.

Aprendeu algumas soluções comuns para problemas recorrentes estudando alguns *Design Patterns*.

Figura 2.1: Exemplo do conteúdo do PDF.

Há diversos bons tutoriais sobre Markdown na Internet. Uma boa introdução é o seguinte artigo: https://www.alura.com.br/artigos/criando-anotacoes-com-markdown

2.2 Um pouco sobre a implementação do Cotuba

O Cotuba foi implementado em Java 17 e usa o Maven como ferramenta de build. A estrutura do código do Cotuba é a seguinte:

```
cotuba
├── pom.xml
│
└── src
    ├── assembly
    │   └── distribution.xml
    ├── main
    │    └── java
    │        └── cotuba
    │             └── Main.java
    │
    └── scripts
         └── cotuba.sh
```

São usadas as seguintes bibliotecas, declaradas como dependências no `pom.xml`:

- *Apache Commons CLI* para as opções de linha de comando: https://commons.apache.org/proper/commons-cli/
- *CommonMark Java* para renderizar arquivos `.md` para HTML: https://github.com/atlassian/commonmark-java
- *iText pdfHTML* para transformar os arquivos HTML em um `.pdf`: https://github.com/itext/i7j-pdfhtml/
- *Epublib* para transformar os arquivos HTML em um `.epub`: https://github.com/psiegman/epublib

O `.zip`, que é o "entregável" do Cotuba, é gerado pelo *Maven Assembly Plugin* (http://maven.apache.org/plugins/maven-assembly-plugin/). A configuração do *Assembly Plugin* está em `src/assembly/distribution.xml`. Nesse XML, é descrito que o `.zip`

conterá os JARs de todas as bibliotecas e o arquivo
`src/script/cotuba.sh` .

2.3 Gerando um ebook

Vamos baixar o projeto do Cotuba em nossa máquina, fazer o build do projeto com o Maven e, em seguida, gerar um ebook em PDF e em EPUB a partir dos arquivos `.md` do diretório `livro-exemplo`. Para isso, primeiramente, vamos abrir um terminal e ir até o *Desktop*:

```
cd ~/Desktop
```

Então, vamos baixar o código do Cotuba usando o Git:

```
git clone
    https://github.com/alexandreaquiles/solid-na-pratica.git
```

Logo após, devemos entrar na pasta do projeto:

```
cd solid-na-pratica/cotuba
```

A seguir, vamos fazer o build do projeto usando o Maven:

```
mvn clean package
```

Deve ser gerado, no diretório `target`, um arquivo `cotuba-cli-0.0.1-SNAPSHOT-distribution.zip`. Precisamos descompactar o `.zip` gerado para o Desktop com o comando:

```
unzip -o target/cotuba-*-distribution.zip -d ~/Desktop
```

Deve ser exibido algo como:

```
Archive:  target/cotuba-cli-0.0.1-SNAPSHOT-distribution.zip
inflating: .../Desktop/cotuba.sh
inflating: .../Desktop/libs/layout-7.1.0.jar
inflating: .../Desktop/libs/commons-cli-1.4.jar
inflating: .../Desktop/libs/commonmark-0.11.0.jar
inflating: .../Desktop/libs/epublib-core-3.1.jar
inflating: .../Desktop/libs/slf4j-api-1.6.1.jar
inflating: .../Desktop/libs/cotuba-cli-0.0.1-SNAPSHOT.jar
```

```
inflating: .../Desktop/libs/slf4j-simple-1.6.1.jar
inflating: .../Desktop/libs/kernel-7.1.0.jar
inflating: .../Desktop/libs/kxml2-2.5.1-20180703.211101-1-jar-with-
dependencies.jar
inflating: .../Desktop/libs/io-7.1.0.jar
inflating: .../Desktop/libs/html2pdf-2.0.0.jar
```

Depois do `.zip` ser descompactado, vamos voltar ao Desktop:

```
cd ~/Desktop
```

Para gerar um PDF do livro de exemplo que já vem com o Cotuba, devemos fazer:

```
./cotuba.sh -d solid-na-pratica/cotuba/livro-exemplo -f pdf
```

Também é possível gerar um EPUB desse livro de exemplo:

```
./cotuba.sh -d solid-na-pratica/cotuba/livro-exemplo -f epub
```

Pronto! Ebooks nos formatos PDF e EPUB gerados!

2.4 O código do Cotuba

O código do Cotuba possui apenas uma classe: a classe `Main` do pacote `cotuba`, que define o método `main` e possui 246 linhas de código, considerando as linhas em branco. Dessas linhas, 40 são declarações de import e do pacote. Esse nosso ponto de partida está listado no capítulo 14 - *Apêndice: O código inicial do Cotuba* e pode ser encontrado no repositório do GitHub já mencionado: https://github.com/alexandreaquiles/solid-na-pratica

> É importante fazer uma ressalva semelhante à de David Parnas no clássico artigo *On the Criteria To Be Used in Decomposing Systems into Modules* que, ao apresentar o exemplo a ser debatido, diz: *"Este é um sistema pequeno. Exceto em circunstâncias extremas [...] tal sistema poderia ser produzido por um bom programador dentro de uma ou duas semanas. Consequentemente, nenhuma das dificuldades que motivam a programação modular é importante para este sistema. Como é impraticável tratar um grande sistema minuciosamente, devemos passar pelo exercício de tratar esse problema como se fosse um grande projeto."* (PARNAS, 1972).

É uma quantidade razoável de código em apenas uma classe. Será que esse código está fácil de ser mantido?

No capítulo seguinte, vamos estudar esse código a fundo e verificar se um dos fundamentos do SOLID é atendido: o Princípio da Responsabilidade Única.

Figura 2.2: Vídeo do capítulo

CAPÍTULO 3

Princípio da Responsabilidade Única (SRP): classes coesas

Vamos dizer que, em um projeto da sua empresa, você se depara com a seguinte classe:

```
public class Empregado {

  public BigDecimal calculaPagamento() {
    //...
  }

  public BigDecimal calculaTaxas() {
    //...
  }

  public BigDecimal calculaHorasExtras(List<Hora> horas) {
    //...
  }


  public void salva() {
    // persiste no Banco de Dados...
  }

  public static Empregado buscaPorId(Long id) {
    // busca do Banco de Dados ...
  }


  public String convertePraXML() {
    //...
  }

  public static Empregado leXML(String xml) {
    //...
  }
}
```

Este exemplo é inspirado no livro *UML for Java Programmers* de Robert C. Martin (MARTIN, 2003).

Como você pode ver no código acima, a classe `Empregado` realiza as seguintes tarefas:

- Calcula o pagamento do empregado;
- Calcula taxas referentes ao empregado;
- Calcula as horas extras do empregado;
- Salva o empregado no Banco de Dados;
- Busca o empregado do Banco de Dados pelo id;
- Converte o empregado para um arquivo XML;
- Lê o empregado de um arquivo XML.

Olhando para essa classe e para as tarefas que ela executa, você diria que ela tem um bom design?

Para analisarmos se essa classe tem um bom design, podemos agrupar essas tarefas em alguns conjuntos de "interessados", tanto relacionados ao negócio como à infraestrutura técnica:

- O cálculo de pagamento e taxas interessa ao **setor Financeiro**.
- O cálculo de horas extras interessa ao **setor de RH**.
- Salvar no banco de dados e buscar por `id` interessa à **implementação de persistência**.
- Converter de/para XML interessa à **integração com outros sistemas**.

Como a classe `Empregado` tem muitos "interessados", ela pode ser afetada por diversas mudanças: nos impostos; nas regras de horas extras; nos nomes de colunas no Banco de Dados; no schema do XML (`.xsd`).

Cada "interessado", de negócio ou técnico, faz com que a classe tenha uma responsabilidade diferente.

Voltando à pergunta feita anteriormente, sob o ponto de vista das *responsabilidades*, podemos dizer que temos um design *ruim*. A classe `Empregado` tem **muitas responsabilidades**.

À medida que um software vai sendo desenvolvido, precisamos alterar o seu código, tanto por motivos técnicos como por motivos relacionados ao negócio. Cada responsabilidade de uma classe é um motivo diferente para mudarmos seu código. Portanto, muitas responsabilidades são **muitas razões para a classe ser modificada**.

3.1 O Princípio da Responsabilidade Única (SRP)

Pensando na distribuição de responsabilidades entre classes, Uncle Bob cunhou o seguinte princípio:

> **SINGLE RESPONSIBILITY PRINCIPLE (SRP)**
>
> *Uma classe deve ter um, e apenas um, motivo para ser modificada.*

Responsabilidade Única não é o mesmo que fazer apenas uma coisa

No livro *Clean Architecture* (MARTIN, 2017), Uncle Bob confessa que o termo *Responsabilidade Única* não foi uma boa escolha de palavras. Classes não necessariamente precisam fazer apenas uma coisa. Isso é uma característica dos métodos (ou funções). A questão principal do SRP é o motivo para uma classe ser modificada. E esse motivo para mudança, em geral, está relacionado a um grupo de usuários ou *stakeholders*, que Uncle Bob chama de *atores*.

Por isso, Uncle Bob define o SRP de maneira um pouco diferente: *"Um módulo deve ser responsável por um, e apenas um, ator."*

A origem do SRP

O foco em organizar o código em torno dos motivos para modificá-lo ecoa as ideias do já mencionado artigo seminal de David Parnas: *On the Criteria To Be Used in Decomposing Systems into Modules* (PARNAS, 1972). No artigo, Parnas descreve dois critérios de decomposição do código: o primeiro, mais comum à época, decompõe o código em torno das etapas de processamento (o *flowchart*, ou fluxograma); o segundo, proposto no artigo, é caracterizado por revelar o mínimo das decisões internas de um módulo para os outros módulos, o que Parnas chama de (*information hiding*, ou ocultação da informação). Em seguida, são listadas algumas possíveis mudanças no problema proposto e é demonstrado que o código decomposto de acordo com o critério de *information hiding* seria menos impactado por essas mudanças. Parnas conclui: *comece com uma lista de decisões de design difíceis ou que são propensas a mudar. Cada módulo é então projetado para ocultar tais decisões dos outros.*

No artigo *The Single Responsibility Principle* (MARTIN, 2014a), Uncle Bob cita o trabalho de Parnas como inspiração para o SRP.

3.2 SRP e classes coesas

Em seu artigo *Cohesion* (VANDERBURG, 2011), Glenn Vanderburg contrasta dois termos que usamos ao nos referirmos à união de duas coisas:

- **adesão**, quando algo externo, como uma cola ou fita crepe, gruda coisas não relacionadas;
- **coesão** quando a união é natural, como dois pedaços de argila ou duas peças de uma máquina.

Vanderburg afirma que uma ideia é coesa (ou seja, tem coesão) quando uma linha de raciocínio é composta por pensamentos que se encaixam, se relacionam e que caminham juntos.

Classes coesas têm uma característica semelhante: os conceitos que essas classes representam estariam relacionados e separá-los seria pouco natural. O SRP, no fim das contas, é uma outra maneira de falar sobre a necessidade de código coeso.

> **Coesão**
>
> *FIG. Coerência de pensamento ou de um todo.*
>
> Fonte: Michaelis Online.

No livro *OO e SOLID para Ninjas* (ANICHE, 2015), Maurício Aniche sugere que, para encontrar classes pouco coesas, devemos procurar classes que:

- Possuem muitos métodos diferentes.
- São modificadas com frequência.
- Não param nunca de crescer.

Ao implementar código novo, nem sempre conseguimos criar classes coesas, que seguem o SRP. A melhor abordagem é sempre revisar seu código, buscando problemas de coesão e corrigindo-os aos poucos.

Métrica: LCOM

Como mencionamos anteriormente, o SRP está relacionado à coesão das classes. No artigo *A metrics suite for object oriented design* (CHIDAMBER; KEMERER, 1994), Shyam Chidamber e Chris Kemerer definiram um conjunto de métricas focadas em código OO.

Entre as métricas descritas está a **LCOM (Lack of Cohesion in Methods)**. A LCOM mede o grau em que os métodos de uma classe acessam todos os atributos. De acordo com essa métrica, uma classe totalmente coesa (cujo LCOM é 0) teria todos os métodos acessando todos os atributos. Quanto menos coesa, maior o valor do LCOM.

A ideia é que um método que acessa apenas partes dos atributos de uma dada classe poderia ser definido em uma nova classe. Os atributos e o métodos seriam movidos para a classe criada e o LCOM seria minimizado.

Essa métrica não deve ser usada de maneira isolada das outras métricas de Chidamber e Kemerer. Pense no nosso ponto de partida, a classe `Main` do Cotuba, que lê as opções da linha de comando, renderiza arquivos Markdown para HTML, gera PDFs e EPUBs. Não podemos chamá-la de coesa, não é mesmo? Mas o LCOM seria 0, o valor mínimo, já que ela tem apenas um método e não tem nenhum atributo, denotando uma classe totalmente coesa.

Por outro lado, pense em uma entidade, que definimos corriqueiramente em nossos projetos. A classe possuiria atributos privados e métodos de acesso (*getters* e *setters*). Teria um LCOM altíssimo, já que os seus métodos acessariam apenas um atributo. Mas não poderíamos dizer que a classe seria pouco coesa.

A métrica LCOM pode fazer sentido para um Controller que recebe várias dependências em seus atributos mas cada método acessa apenas uma delas. Tal Controller apresentaria alto LCOM, indicando que poderia ser quebrado em classes mais coesas e, em consequência, com responsabilidades melhor distribuídas.

Alguns autores sugerem versões alternativas da métrica, chamadas de LCOM2 e LCOM3.

3.3 Responsabilidades e SRP no Cotuba

No capítulo anterior, mencionamos que o código do Cotuba é composto por apenas uma classe `Main` de 246 linhas de código. É provável que `Main` tenha mais de um motivo para ser modificada. Para analisarmos se essa classe atende ao SRP, precisamos responder a duas perguntas:

- quais as responsabilidades dessa classe?
- quais os motivos para essa classe ser modificada?

Em termos de responsabilidades, a classe `Main` realiza as seguintes tarefas:

- Lê as opções da linha de comando.
- Renderiza arquivos `.md` para HTML.
- Gera um ebook no formato `.pdf`.
- Gera um ebook no formato `.epub`.

Pensando em motivos para mudanças, cada detalhe relacionado a essas responsabilidades poderia levar a uma necessidade de modificação da classe `Main`.

Ouvindo os imports

Uma maneira de avaliar se uma classe adere ao SRP é verificar se suas dependências são coerentes entre si. Se as dependências não tiverem uma relação que faça sentido, é provável que tenhamos uma classe com muitas responsabilidades diferentes.

Em Java, para analisar as dependências de uma classe, podemos observar seus imports. Se os imports forem de pacotes de bibliotecas sem relação alguma, é possível que haja uma quebra do SRP.

No livro *Growing Object-Oriented Software, Guided by Tests* (FREEMAN; PRYCE, 2009), Steve Freeman e Nat Pryce analisam uma classe que importa código de outros três pacotes que lidam com telas Desktop (Swing/AWT), com o protocolo de comunicação XMPP e com a aplicação de leilões que é desenvolvida durante o livro. O fato de os pacotes importados não estarem relacionados entre si faz com que os autores considerem que a classe analisada possui responsabilidades demais.

Na classe `Main` do Cotuba, há imports dos pacotes:

- `org.apache.commons.cli`, da biblioteca Apache Commons CLI, para ler as opções da linha de comando.
- `org.commonmark`, da biblioteca CommonMark Java, para renderizar arquivos Markdown para HTML.
- `com.itextpdf`, da biblioteca iText, para gerar ebooks no formato PDF.
- `nl.siegmann.epublib`, da biblioteca Epublib, para gerar ebooks no formato EPUB.
- `java.io`, `java.nio` para manipular arquivos.
- `java.util` e `java.util.stream` para manipular listas.

Sem dúvida, a classe `Main` fere o SRP. Para fazer com que o código do Cotuba siga o SRP, precisamos mover algumas das responsabilidades da classe `Main` para outras classes. Uma ideia inicial é criar as seguintes novas classes:

- Uma classe para ler as opções de linha de comando, que usará classes do pacote `org.apache.commons.cli`.
- Uma classe para gerar PDFs, que usará classes do pacote `com.itextpdf`.
- Uma classe para gerar EPUBs, que usará classes do pacote `nl.siegmann.epublib`.

Uma classe para ler as opções de linha de comando

Vamos iniciar nossa jornada rumo ao SRP no Cotuba criando uma classe responsável por ler as opções de linha de comando chamada `LeitorOpcoesCLI`. Em seguida, vamos utilizar a nova classe na `Main`. Depois das mudanças, os imports do pacote `org.apache.commons.cli`, referentes à biblioteca Apache Commons CLI, serão encontrados apenas nessa nova classe.

Nosso pontapé inicial é criar uma nova classe `LeitorOpcoesCLI` dentro do pacote `cotuba`, definindo um construtor que recebe como parâmetro `String[] args`:

```
// cotuba.LeitorOpcoesCLI

package cotuba;

public class LeitorOpcoesCLI {

  public LeitorOpcoesCLI(String[] args) {

  }

}
```

Então, vamos mover o seguinte código da classe `Main` para dentro do construtor de `LeitorOpcoesCLI`:

```
// cotuba.LeitorOpcoesCLI

var options = new Options();

var opcaoDeDiretorioDosMD = new Option("d", "dir", true,
    "Diretório que contém os arquivos md."
        + " Default: diretório atual.");
options.addOption(opcaoDeDiretorioDosMD);

var opcaoDeFormatoDoEbook = new Option("f", "format", true,
    "Formato de saída do ebook. Pode ser: pdf ou epub."
        + " Default: pdf");
options.addOption(opcaoDeFormatoDoEbook);
```

```
var opcaoDeArquivoDeSaida = new Option("o", "output", true,
    "Arquivo de saída do ebook. Default: book.{formato}.");
options.addOption(opcaoDeArquivoDeSaida);

var opcaoModoVerboso = new Option("v", "verbose", false,
    "Habilita modo verboso.");
options.addOption(opcaoModoVerboso);

CommandLineParser cmdParser = new DefaultParser();
var ajuda = new HelpFormatter();
CommandLine cmd;

try {
  cmd = cmdParser.parse(options, args);
} catch (ParseException e) {
  System.err.println(e.getMessage());
  ajuda.printHelp("cotuba", options);
  System.exit(1);
  return;
}
```

Não podemos esquecer de fazer os imports relativos às classes do pacote `org.apache.commons.cli` .

> Durante as alterações deste capítulo, devem acontecer alguns erros de compilação. Não se preocupe! Vamos corrigi-los aos poucos.

Nosso próximo passo é mover o trecho a seguir de `Main` para o finalzinho do construtor de `LeitorOpcoesCLI` :

```
// cotuba.LeitorOpcoesCLI

// código omitido...

String nomeDoDiretorioDosMD = cmd.getOptionValue("dir");

if (nomeDoDiretorioDosMD != null) {
  diretorioDosMD = Paths.get(nomeDoDiretorioDosMD);
```

```
  if (!Files.isDirectory(diretorioDosMD)) {
    throw new IllegalArgumentException(nomeDoDiretorioDosMD
        + " não é um diretório.");
  }
} else {
  Path diretorioAtual = Paths.get("");
  diretorioDosMD = diretorioAtual;
}

String nomeDoFormatoDoEbook = cmd.getOptionValue("format");

if (nomeDoFormatoDoEbook != null) {
    formato = nomeDoFormatoDoEbook.toLowerCase();
} else {
  formato = "pdf";
}

String nomeDoArquivoDeSaidaDoEbook =
    cmd.getOptionValue("output");
if (nomeDoArquivoDeSaidaDoEbook != null) {
  arquivoDeSaida = Paths.get(nomeDoArquivoDeSaidaDoEbook);
} else {
  arquivoDeSaida = Paths.get("book." + formato.toLowerCase());
}
if (Files.isDirectory(arquivoDeSaida)) {
  // deleta arquivos do diretório recursivamente
  Files.walk(arquivoDeSaida).sorted(Comparator.reverseOrder())
      .map(Path::toFile).forEach(File::delete);
} else {
  Files.deleteIfExists(arquivoDeSaida);
}

modoVerboso = cmd.hasOption("verbose");
```

Não podemos deixar de fazer os imports das classes do pacote `java.nio.file`, `java.io` e `java.util`. O início da classe `Main` deve ficar assim:

```
// cotuba.Main

public class Main {
```

```
  public static void main(String[] args) {

    Path diretorioDosMD;
    String formato;
    Path arquivoDeSaida;
    boolean modoVerboso = false;

    try {

      if ("pdf".equals(formato)) {

        //restante do código...
```

Nesse momento, vários erros de compilação devem estar acontecendo tanto na classe `LeitorOpcoesCLI` como na `Main`. Vamos corrigir os erros de `LeitorOpcoesCLI` envolvendo o conteúdo do construtor em um bloco *try/catch* que relança uma possível `IOException` como a exceção não checada `IllegalArgumentException` e declarando os seguintes atributos:

```
// cotuba.LeitorOpcoesCLI

public class LeitorOpcoesCLI {

  private Path diretorioDosMD; // inserido
  private String formato; // inserido
  private Path arquivoDeSaida; // inserido
  private boolean modoVerboso = false; // inserido

  public LeitorOpcoesCLI(String[] args) {

    try { // inserido

      // código omitido...

    } catch (IOException ex) { // inserido
      throw new IllegalArgumentException(ex);
    }
```

```
  }
}
```

Agora, vamos gerar os *getters* para esses atributos. Os *setters* NÃO serão necessários. Para digitar menos, podemos usar nossa IDE!

Em seguida, vamos modificar o início do método `main` da classe `Main`, instanciando `LeitorOpcoesCLI` e usando os getters para obter os valores das opções de geração do ebook:

```
// cotuba.Main

public class Main {

  public static void main(String[] args) {

    Path diretorioDosMD;
    String formato;
    Path arquivoDeSaida;
    boolean modoVerboso = false;

    try {

      var opcoesCLI = new LeitorOpcoesCLI(args); //inserido

      diretorioDosMD = opcoesCLI.getDiretorioDosMD(); // inserido
      formato = opcoesCLI.getFormato(); // inserido
      arquivoDeSaida = opcoesCLI.getArquivoDeSaida(); // inserido
      modoVerboso = opcoesCLI.isModoVerboso(); // inserido

      //restante do código...
```

Depois dessa alteração, vários imports desnecessários, como os do pacote `org.apache.commons.cli`, podem ser removidos da classe `Main`. A IDE é um grande auxílio nessa tarefa!

Os imports da biblioteca Apache Commons CLI agora estão isolados na classe responsável por extrair as opções de linha de comando, a `LeitorOpcoesCLI`. O código do Cotuba está tomando

forma! Agora, podemos gerar um PDF e/ou EPUB novamente. Tudo deve funcionar corretamente!

Uma classe para gerar PDF

Continuando nossa jornada rumo ao SRP no Cotuba, extrairemos da `Main` uma nova classe responsável por gerar PDFs. Todo os imports do pacote `com.itextpdf`, relativos à biblioteca de geração de PDFs iText, devem ser encontrados somente nessa nova classe.

Vamos criar uma nova classe chamada `GeradorPDF`, dentro do pacote `cotuba`, definindo um método `gera` que recebe os parâmetros `diretorioDosMD` e `arquivoDeSaida`, ambos do tipo `Path`:

```
// cotuba.GeradorPDF

package cotuba;

import java.nio.file.Path;

public class GeradorPDF {

  public void gera(Path diretorioDosMD, Path arquivoDeSaida) {

  }

}
```

Então, vamos mover código a seguir, que efetua a geração do PDF, de dentro do `if ("pdf".equals(formato))` da classe `Main` para o método `gera` de `GeradorPDF`:

```
// cotuba.GeradorPDF

try (var writer = new PdfWriter(
          Files.newOutputStream(arquivoDeSaida));
  var pdf = new PdfDocument(writer);
  var pdfDocument = new Document(pdf)) {

  PathMatcher matcher = FileSystems.getDefault()
```

```
    .getPathMatcher("glob:**/*.md");
try (Stream<Path> arquivosMD = Files.list(diretorioDosMD)) {
  arquivosMD
    .filter(matcher::matches)
    .sorted()
    .forEach(arquivoMD -> {
      Parser parser = Parser.builder().build();
      Node document = null;
      try {
        document = parser.parseReader(
              Files.newBufferedReader(arquivoMD));
        document.accept(new AbstractVisitor() {
          @Override
          public void visit(Heading heading) {
            if (heading.getLevel() == 1) {
              // capítulo
              String tituloDoCapitulo = ((Text) heading
                  .getFirstChild()).getLiteral();
              // TODO: usar título do capítulo
            } else if (heading.getLevel() == 2) {
              // seção
            } else if (heading.getLevel() == 3) {
              // título
            }
          }

        });
      } catch (Exception ex) {
        throw new IllegalStateException(
            "Erro ao fazer parse do arquivo "
                + arquivoMD, ex);
      }

      try {
        HtmlRenderer renderer = HtmlRenderer.builder().build();
        String html = renderer.render(document);

        List<IElement> convertToElements =
            HtmlConverter.convertToElements(html);
        for (IElement element : convertToElements) {
          pdfDocument.add((IBlockElement) element);
```

```
            }
            // TODO: não adicionar página depois do último capítulo
            pdfDocument.add(
                  new AreaBreak(AreaBreakType.NEXT_PAGE));

          } catch (Exception ex) {
            throw new IllegalStateException(
              "Erro ao renderizar para HTML o arquivo "
                  + arquivoMD, ex);
          }

        });
    } catch (IOException ex) {
      throw new IllegalStateException(
        "Erro tentando encontrar arquivos .md em "
            + diretorioDosMD.toAbsolutePath(), ex);
    }

} catch (Exception ex) {
    throw new IllegalStateException(
      "Erro ao criar arquivo PDF: "
          + arquivoDeSaida.toAbsolutePath(), ex);
}
```

Não podemos esquecer de adicionar, em `GeradorPDF`, os imports necessários dos pacotes `org.commonmark`, `com.itextpdf`, além dos referentes às bibliotecas padrão do Java.

Agora, podemos usar `GeradorPDF` na classe `Main`:

```
// cotuba.Main

    if ("pdf".equals(formato)) {

      // modificado
      var geradorPDF = new GeradorPDF();
      geradorPDF.gera(diretorioDosMD, arquivoDeSaida);

    } else if ("epub".equals(formato)) {

    // restante do código...
```

Claro, não devemos esquecer de limpar os imports desnecessários da classe `Main`. A geração de PDFs deve continuar funcionando!

Uma classe para gerar EPUB

A próxima etapa da nossa jornada em direção ao SRP no Cotuba será criar uma classe responsável por gerar EPUBs, usando-a na `Main`. Os imports do pacote `nl.siegmann.epublib`, referente à biblioteca de geração de EPUBs Epublib, devem ser necessários apenas nessa nova classe.

Vamos criar uma nova classe `GeradorEPUB` dentro do pacote `cotuba`, definindo um método `gera` com os parâmetros `diretorioDosMD` e `arquivoDeSaida`, ambos do tipo `Path`:

```
// cotuba.GeradorEPUB

package cotuba;

import java.nio.file.Path;

public class GeradorEPUB {

  public void gera(Path diretorioDosMD, Path arquivoDeSaida) {

  }

}
```

O trecho da classe `Main` de dentro do `if ("epub".equals(formato))`, que gera EPUBs, deve ser movido para o método `gera` de `GeradorEPUB`:

```
// cotuba.GeradorEPUB

var epub = new Book();

PathMatcher matcher = FileSystems.getDefault()
    .getPathMatcher("glob:**/*.md");
try (Stream<Path> arquivosMD = Files.list(diretorioDosMD)) {
```

```
arquivosMD
  .filter(matcher::matches)
  .sorted()
    .forEach(arquivoMD -> {
      Parser parser = Parser.builder().build();
      Node document = null;
      try {
        document = parser.parseReader(
              Files.newBufferedReader(arquivoMD));
        document.accept(new AbstractVisitor() {
          @Override
          public void visit(Heading heading) {
            if (heading.getLevel() == 1) {
              // capítulo
              String tituloDoCapitulo = ((Text) heading
                  .getFirstChild()).getLiteral();
              // TODO: usar título do capítulo
            } else if (heading.getLevel() == 2) {
              // seção
            } else if (heading.getLevel() == 3) {
              // título
            }
          }

        });
      } catch (Exception ex) {
        throw new IllegalStateException(
          "Erro ao fazer parse do arquivo "
              + arquivoMD, ex);
      }

      try {
        HtmlRenderer renderer = HtmlRenderer.builder().build();
        String html = renderer.render(document);

        // TODO: usar título do capítulo
        epub.addSection("Capítulo",
            new Resource(html.getBytes(),
                MediatypeService.XHTML));

      } catch (Exception ex) {
```

```
          throw new IllegalStateException(
            "Erro ao renderizar para HTML o arquivo "
                + arquivoMD, ex);
        }
      });

} catch (IOException ex) {
  throw new IllegalStateException(
    "Erro tentando encontrar arquivos .md em "
        + diretorioDosMD.toAbsolutePath(), ex);
}

var epubWriter = new EpubWriter();

try {
  epubWriter.write(epub, Files.newOutputStream(arquivoDeSaida));
} catch (IOException ex) {
  throw new IllegalStateException(
    "Erro ao criar arquivo EPUB: "
        + arquivoDeSaida.toAbsolutePath(), ex);
}
```

Os imports corretos devem ser adicionados referentes aos pacotes `org.commonmark`, `nl.siegmann.epublib` e às bibliotecas padrão do Java.

Devemos usar a nova classe `GeradorEPUB` em `Main`. O trecho de `Main` que gera ebooks deverá ficar parecido com:

```
// cotuba.Main

  // código omitdo ...

  } else if ("epub".equals(formato)) {

    // modificado
    var geradorEPUB = new GeradorEPUB();
    geradorEPUB.gera(diretorioDosMD, arquivoDeSaida);

  } else {

  // restante do código...
```

Agora, podemos remover os imports desnecessários de `Main`. O único import que restará em `Main` será o da classe `java.nio.file.Path`! Ao testarmos a geração de EPUBs, tudo deve funcionar.

Refatorar é bom para aprender OO

Depois das alterações, extraímos três classes de `Main`: as classes `LeitorOpcoesCLI`, `GeradorPDF` e `GeradorEPUB`.

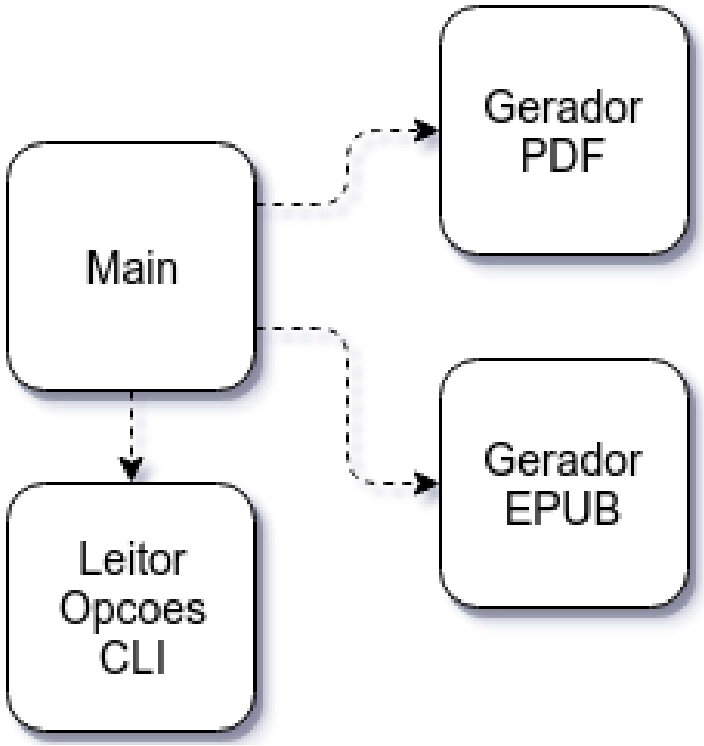

Figura 3.1: Classes com responsabilidades definidas.

As responsabilidades ficaram mais bem distribuídas e o código, mais fácil de entender e manter. E, claro, mais aderente ao SRP, já que cada uma das classes tem responsabilidades mais definidas e menos motivos para serem modificadas. Note que melhoramos o código sem inserir novas funcionalidades ou mudar as já existentes: fizemos uma **refatoração**.

> *"Refatoração (substantivo): uma alteração feita na estrutura interna do software para torná-lo mais fácil de ser entendido e menos custoso de ser modificado sem alterar seu comportamento observável."*
>
> Martin Fowler, no livro *Refactoring* (FOWLER et al., 1999)

Começar de um código bagunçado, com muitas responsabilidades, modificá-lo em pequenos passos até chegar a um código extensível e com responsabilidades bem definidas é uma boa maneira de ver a verdadeira utilidade de OO. E de aprender!

3.4 Não se repita

Código duplicado, triplicado, quadruplicado, enfim, código repetido, é algo muito comum. E muito ruim. Quando temos código repetido, uma mudança nas regras de negócio ou na tecnologia pode requerer uma alteração em vários pontos do código. O esforço de manutenção é grande. Além disso, há a possibilidade de esquecermos de alterar algum ponto e, em consequência, gerarmos um *bug* no nosso sistema.

Mas como evitar código repetido? Extraindo a lógica comum para um outro método ou outra classe. No fim das contas, quando temos repetição de código, há algo novo pedindo para nascer. E esse novo código terá, provavelmente, uma responsabilidade bem definida.

Os "Programadores Pragmáticos" Andy Hunt e Dave Thomas criaram, no livro *Pragmatic Programmer* (HUNT; THOMAS, 1999), um acrônimo que serve como um mantra para bons programadores: D.R.Y. ou **Don't Repeat Yourself** (não se repita): *"Todo bloco de conhecimento deve ter uma representação única, sem ambiguidades e dominante num sistema."*

> *"Dependência é o principal problema em desenvolvimento de software. Duplicação é o sintoma. Mas, ao contrário da maioria dos problemas na vida, nos quais eliminar os sintomas faz com que um problema mais grave apareça em outro lugar, eliminar duplicação nos programas elimina a dependência."*
>
> Kent Beck, no livro *TDD by Example* (BECK, 2002)

Quando refatorar algo repetido no código?

Martin Fowler, no livro *Refactoring* (FOWLER et al., 1999), cita uma diretriz que recebeu de Don Roberts, que chama de **a regra dos três**:

"A primeira vez que você faz algo, apenas faça. Na segunda vez que você faz algo parecido, incomode-se com a duplicação, mas duplique de qualquer forma. Na terceira vez que você faz algo semelhante, refatore."

Duplicação no Cotuba

Você percebeu que tanto a classe `GeradorPDF` como a classe `GeradorEPUB` têm vários imports em comum?

Em ambos os geradores, há imports de classes dos pacotes `java.io`, `java.nio` e `java.util`. Até aí tudo bem. É apenas um sinal de algo estranho... Mas o problema é que boa parte desses imports em comum são do pacote `org.commonmark`, da biblioteca CommonMark Java, que serve para renderizar arquivos Markdown para HTML.

Será que é responsabilidade dos geradores de PDF e EPUB transformar arquivos `.md` em HTML? Acho que há uma classe pedindo para nascer... E essa nova classe reforçará o SRP, já que terá uma responsabilidade bem definida: a renderização de arquivos `.md` para HTML.

Extraindo duplicação para uma classe renderizadora de Markdown

Continuando nossa jornada de refatoração do Cotuba, vamos criar uma classe responsável por renderizar arquivos `.md` para HTML. Os imports do pacote `org.commonmark`, referentes à biblioteca CommonMark Java, devem ficar apenas nessa nova classe.

Primeiramente, vamos criar uma classe chamada `RenderizadorMDParaHTML` no pacote `cotuba`. Nessa nova classe, vamos

definir o método `renderiza`, que receberá como parâmetro o `Path` do diretório que contém os arquivos `.md`:

```
// cotuba.RenderizadorMDParaHTML

package cotuba;

import java.nio.file.Path;

public class RenderizadorMDParaHTML {

  public void renderiza(Path diretorioDosMD) {

  }

}
```

Logo após, vamos mover o trecho a seguir, que está duplicado em `GeradorPDF` e `GeradorEPUB`, para o método `renderiza` de `RenderizadorMDParaHTML`:

```
// cotuba.RenderizadorMDParaHTML

PathMatcher matcher = FileSystems.getDefault()
    .getPathMatcher("glob:**/*.md");
try (Stream<Path> arquivosMD = Files.list(diretorioDosMD)) {
  arquivosMD
    .filter(matcher::matches)
    .sorted()
    .forEach(arquivoMD -> {

      Parser parser = Parser.builder().build();
      Node document = null;
      try {
        document = parser.parseReader(
              Files.newBufferedReader(arquivoMD));
        document.accept(new AbstractVisitor() {
          @Override
          public void visit(Heading heading) {
            if (heading.getLevel() == 1) {
              // capítulo
```

```
                String tituloDoCapitulo = ((Text) heading
                    .getFirstChild()).getLiteral();
                // TODO: usar título do capítulo
              } else if (heading.getLevel() == 2) {
                // seção
              } else if (heading.getLevel() == 3) {
                // título
              }
            }

          });

        } catch (Exception ex) {
          throw new IllegalStateException(
            "Erro ao fazer parse do arquivo "
                + arquivoMD, ex);
        }

        try {
          HtmlRenderer renderer = HtmlRenderer.builder().build();
          String html = renderer.render(document);

          // PDF ou EPUB aqui?

        } catch (Exception ex) {
          throw new IllegalStateException(
            "Erro ao renderizar para HTML o arquivo "
                + arquivoMD, ex);
        }

      });
} catch (IOException ex) {
  throw new IllegalStateException(
    "Erro tentando encontrar arquivos .md em "
        + diretorioDosMD.toAbsolutePath(), ex);
}
```

Na classe `RenderizadorMDParaHTML`, devem ser feitos imports de classes do pacote `org.commonmark`, além de `java.io`, `java.nio` e `java.util`.

O código de renderização foi movido para a nova classe. Mas o que devemos fazer com a variável `html`, que contém o conteúdo renderizado de cada arquivo `.md`? Devemos gerar um PDF ou um EPUB? Estudaremos opções mais adiante.

> Na verdade, se olharmos com atenção, a classe `RenderizadorMDParaHTML` tem mais de uma responsabilidade. Essa violação do SRP nos atrapalhará em capítulos posteriores. Resolveremos no momento oportuno!

A classe `GeradorPDF`, depois de esse trecho de código ser movido, deve ter o seguinte conteúdo:

```
// cotuba.GeradorPDF

public class GeradorPDF {

  public void gera(Path diretorioDosMD, Path arquivoDeSaida) {

    try(var writer = new PdfWriter(
          Files.newOutputStream(arquivoDeSaida));
      var pdf = new PdfDocument(writer);
      var pdfDocument = new Document(pdf)) {

      List<IElement> convertToElements =
          HtmlConverter.convertToElements(html);
              // error: cannot find symbol
      for (IElement element : convertToElements) {
        pdfDocument.add((IBlockElement) element);
      }
      // TODO: não adicionar página depois do último capítulo
      pdfDocument.add(new AreaBreak(AreaBreakType.NEXT_PAGE));

    } catch (Exception ex) {
      throw new IllegalStateException(
        "Erro ao criar arquivo PDF: "
            + arquivoDeSaida.toAbsolutePath(), ex);
    }
```

```
  }

}
```

Já a classe `GeradorEPUB` ficará deste jeito:

```
// cotuba.GeradorEPUB

public class GeradorEPUB {

  public void gera(Path diretorioDosMD, Path arquivoDeSaida) {

    var epub = new Book();

    // TODO: usar título do capítulo
    epub.addSection("Capítulo",
        new Resource(html.getBytes(),
          MediatypeService.XHTML)); // error: cannot find symbol

    var epubWriter = new EpubWriter();

    try {
      epubWriter.write(epub,
          Files.newOutputStream(arquivoDeSaida));
    } catch (IOException ex) {
      throw new IllegalStateException(
        "Erro ao criar arquivo EPUB: "
            + arquivoDeSaida.toAbsolutePath(), ex);
    }

  }

}
```

Devemos nos certificar de que removemos os imports desnecessários de `GeradorPDF` e `GeradorEPUB`, em especial de classes do pacote `org.commonmark`.

Por enquanto, vamos ignorar os erros de compilação nas classes geradoras de ebooks. Esses erros estão relacionados à variável

`html`, que deve conter o resultado da renderização de cada arquivo `.md`, tanto no gerador de PDF como no gerador de EPUB. Vamos arrumar a compilação mais à frente.

Estamos quase lá, caminhando cada vez mais na direção do SRP. Mas antes de continuarmos, precisamos avaliar as opções que temos para passar o HTML renderizado para as classes que geram os ebooks.

Avaliando opções de design do código

Precisamos passar, de alguma maneira, o HTML renderizado a partir dos arquivos `.md` para os geradores de PDF e EPUB. Uma questão importante é definir quem chama quem. Há várias implementações possíveis. Entre elas:

- `Main` chama `RenderizadorMDParaHTML`, que, por sua vez, chama `GeradorPDF` e `GeradorEPUB`.

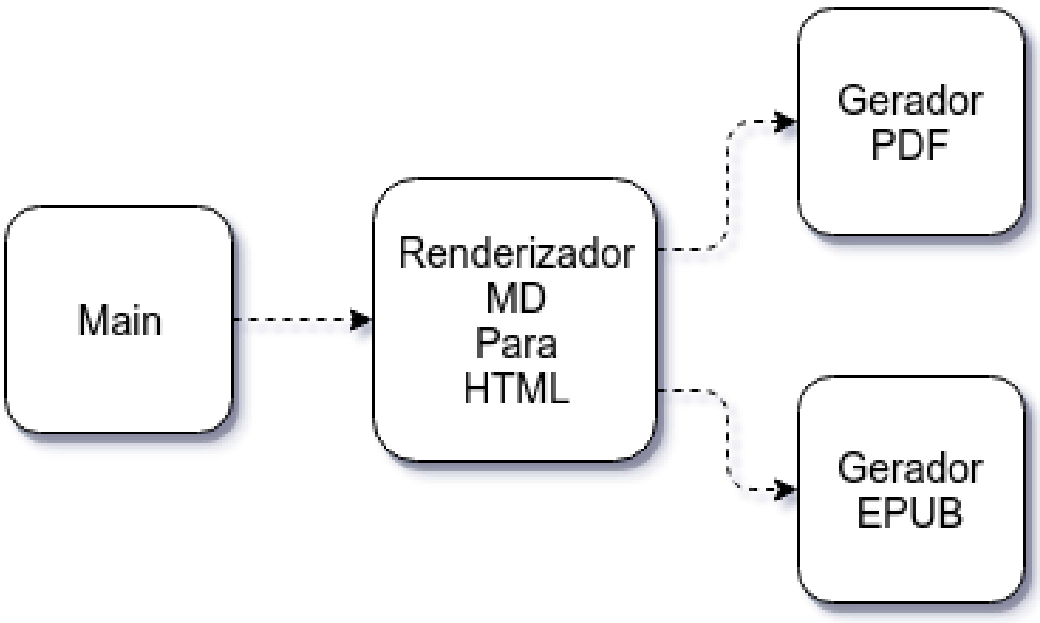

Figura 3.2: Main chama apenas renderizador.

- `Main` chama `RenderizadorMDParaHTML`, `GeradorPDF` e `GeradorEPUB`, coordenando todas as classes.

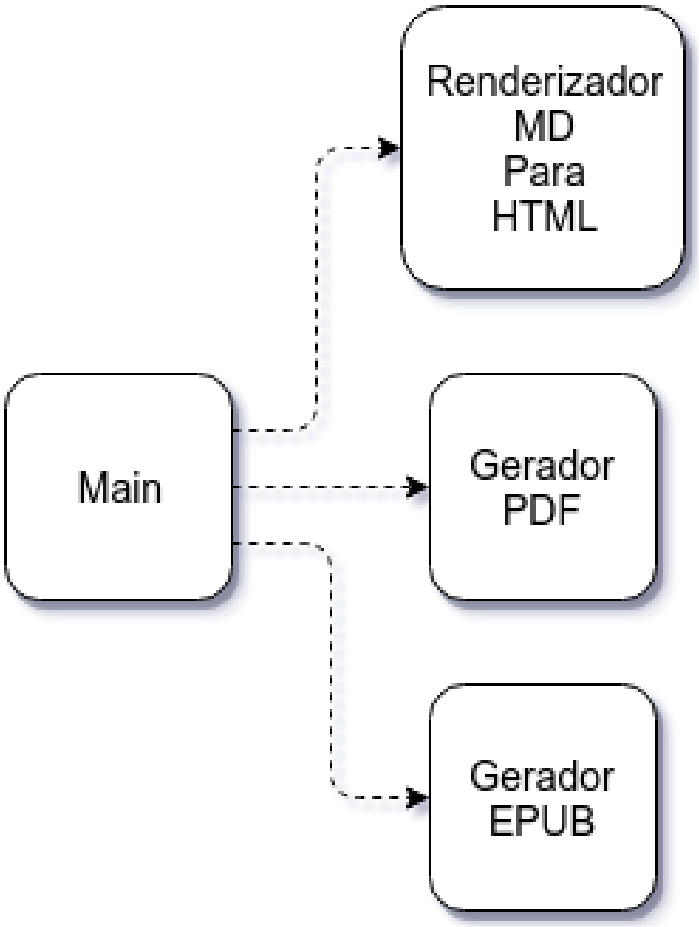

Figura 3.3: Main coordena as classes

Se a classe `RenderizadorMDParaHTML` chamar os geradores, teríamos dois motivos para mudá-la:

- quando tivermos uma mudança na renderização do Markdown (é OK)
- quando acontecer alguma alteração nos geradores (NÃO é OK)

A classe `Main` já é a nossa coordenadora porque:

- colabora com `LeitorOpcoesCLI` para extrair as opções da linha de comando

- colabora com `RenderizadorMDParaHTML` para disparar a renderização do Markdown

Teremos mais duas funções de coordenação na classe `Main`:

- colaborar com `GeradorPDF` para gerar PDFs
- colaborar com `GeradorEPUB` para gerar EPUBs

Pensando em termos de distribuição de responsabilidades, ter a classe `Main` como "coordenadora" é uma opção melhor. Mas como `Main` deve passar o HTML para os geradores? Como uma simples `String`? Por meio de um arquivo?

3.5 Um domain model simples

Uma ótima opção é passar os dados do ebook a ser gerado entre as classes do Cotuba por meio de um modelo de domínio, ou *domain model*: uma representação em código dos conceitos de negócio.

Como o problema de negócio que estamos resolvendo é relacionado à geração de ebooks, temos os seguintes conceitos importantes:

- **ebook**: tem um formato (PDF ou EPUB), um arquivo de saída e uma lista de capítulos
- **capítulo**: tem um título e um conteúdo HTML

O nosso *domain model* é bem simples: teremos uma classe `Ebook` e uma classe `Capitulo`. Só isso mesmo! Organizaremos o código de maneira que a classe `RenderizadorMDParaHTML` retorne uma `List<Capitulo>` para a `Main`. A classe `Main`, por sua vez, cria um `Ebook` contendo a lista de capítulos e os repassa para os geradores de ebook.

Um domain model para a geração de ebooks

Precisamos definir:

- uma classe que represente um capítulo, que contém um título e o conteúdo HTML renderizado;
- uma classe que represente o ebook, que contém um formato, um arquivo de saída e uma lista de capítulos.

Para isso, vamos criar uma classe que `Capitulo`, no pacote `cotuba`, com os atributos privados `titulo` (`String`) e `conteudoHTML` (`String`):

```
// cotuba.Capitulo
```

```
package cotuba;

public class Capitulo {

  private String titulo;

  private String conteudoHTML;

  // getters e setters...

}
```

Com a ajuda de nossa IDE, devemos gerar os *getters* e *setters* para ambos os atributos.

Em seguida, criaremos uma classe `Ebook`, também no pacote `cotuba`, com os atributos privados `formato` (`String`), `arquivoDeSaida` (`Path`) e `capitulos` (`List<Capitulo>`):

```
// cotuba.Ebook

package cotuba;

import java.nio.file.Path;
import java.util.List;

public class Ebook {

  private String formato;

  private Path arquivoDeSaida;

  private List<Capitulo> capitulos;

  // getters e setters...

}
```

Devemos gerar os *getters* e *setters* para os atributos. Não podemos nos esquecer de certificar que os imports estão corretos. Nesse

ponto, as classes `GeradorPDF` e `GeradorEPUB` ainda continuarão com erros de compilação.

Usando o domain model no renderizador

Vamos fazer com que a classe `RenderizadorMDParaHTML` monte uma `List<Capitulo>` e a retorne no método `renderiza`. Para isso, vamos modificar o método `renderiza` de `RenderizadorMDParaHTML`, para que retorne uma `List<Capitulo>`. Devemos instanciar uma lista no começo do método, retornando-a no final:

```
// cotuba.RenderizadorMDParaHTML

// outros imports...

import java.util.ArrayList;
import java.util.List;

public class RenderizadorMDParaHTML {

  ~~public void renderiza(Path diretorioDosMD) {~~
  public List<Capitulo> renderiza(Path diretorioDosMD) {

    List<Capitulo> capitulos = new ArrayList<>(); // inserido

    // código omitido...

    return capitulos; // inserido
  }
}
```

Ainda em `RenderizadorMDParaHTML`, vamos instanciar um `Capitulo` dentro do *for-each*:

```
// cotuba.RenderizadorMDParaHTML

// código omitido...

arquivosMD
  .filter(matcher::matches)
```

```
  .sorted()
  .forEach(arquivoMD -> {

    Capitulo capitulo = new Capitulo(); // inserido

  //restante do código...
```

Então, vamos definir o título do capítulo, removendo o *TODO*:

```
// cotuba.RenderizadorMDParaHTML

// código omitido...

if (heading.getLevel() == 1) {
  // capítulo
  String tituloDoCapitulo = ((Text) heading.getFirstChild())
    .getLiteral();

  //~~TODO: usar título do capítulo~~
  capitulo.setTitulo(tituloDoCapitulo); // inserido

} else if (heading.getLevel() == 2) {

  //restante do código...
```

Em seguida, vamos definir o HTML e adicionar o capítulo à lista instanciada anteriormente:

```
// cotuba.RenderizadorMDParaHTML

// código omitido...

try {
  HtmlRenderer renderer = HtmlRenderer.builder().build();
  String html = renderer.render(document);

  //~~PDF ou EPUB aqui?~~
  capitulo.setConteudoHTML(html); // inserido

  capitulos.add(capitulo); // inserido
```

```
  //restante do código...
```

Os erros de compilação das classes `GeradorPDF` e `GeradorEPUB` ainda continuarão.

Usando o domain model na classe principal e nos geradores de ebook

Na classe `Main`, precisamos chamar o método `renderiza` da classe `RenderizadorMDParaHTML` e montar um `Ebook` com a lista de capítulos retornada. Além disso, devemos passar o `Ebook` para os geradores de PDF e EPUB. Depois disso, devemos usar o `Ebook` nas classes `GeradorPDF` e `GeradorEPUB`.

Em `Main`, vamos criar uma instância de `RenderizadorMDParaHTML` e invocar o método `renderiza`, passando o parâmetro `diretorioDosMD`. O retorno deve ser armazenado em uma variável chamada `capitulos`:

```
// cotuba.Main

try {

  // código omitido...
  modoVerboso = opcoesCLI.isModoVerboso();

  var renderizador = new RenderizadorMDParaHTML(); // inserido
  List<Capitulo> capitulos =
      renderizador.renderiza(diretorioDosMD); // inserido

  if ("pdf".equals(formato)) {
  //restante do código...
```

Ainda em `Main`, devemos criar um objeto `Ebook`, definindo o `formato`, `arquivoDeSaida` e os `capitulos`:

```
// cotuba.Main

// código omitido...
```

```
List<Capitulo> capitulos =
    renderizador.renderiza(diretorioDosMD);

Ebook ebook = new Ebook(); // inserido
ebook.setFormato(formato); // inserido
ebook.setArquivoDeSaida(arquivoDeSaida); // inserido
ebook.setCapitulos(capitulos); // inserido

if ("pdf".equals(formato)) {

//restante do código...
```

Em vez de passar `diretorioDosMD` e `arquivoDeSaida` para o método `gera` dos geradores, vamos passar apenas o `Ebook`:

```
// cotuba.Main
// código omitido...

if ("pdf".equals(formato)) {

  var geradorPDF = new GeradorPDF();
  geradorPDF.gera(diretorioDosMD, arquivoDeSaida);
  geradorPDF.gera(ebook);

} else if ("epub".equals(formato)) {

  var geradorEPUB = new GeradorEPUB();
  geradorEPUB.gera(diretorioDosMD, arquivoDeSaida);
  geradorEPUB.gera(ebook);

} else {

//restante do código...
```

Não podemos nos esquecer de modificar a assinatura do método `gera` de `GeradorPDF` para receber o `Ebook`:

```
// cotuba.GeradorPDF

public class GeradorPDF {
```

```
  public void gera(Path diretorioDosMD, Path arquivoDeSaida) {
  public void gera(Ebook ebook) {
    // código omitido...
  }

}
```

O mesmo deve ser feito em `GeradorEPUB`:

```
// cotuba.GeradorEPUB

public class GeradorEPUB {

  public void gera(Path diretorioDosMD, Path arquivoDeSaida) {
  public void gera(Ebook ebook) {
    // código omitido...
  }

}
```

Dentro do método `gera` de `GeradorPDF`, devemos criar uma variável para armazenar o `arquivoDeSaida` do Ebook e percorrer a lista de capítulos, usando o HTML de cada capítulo na geração do PDF:

```
// cotuba.GeradorPDF

public class GeradorPDF {

  public void gera(Ebook ebook) {

  Path arquivoDeSaida = ebook.getArquivoDeSaida(); // inserido

  try (var writer = new PdfWriter(
        Files.newOutputStream(arquivoDeSaida));
    var pdf = new PdfDocument(writer);
    var pdfDocument = new Document(pdf)){

    for (Capitulo capitulo : ebook.getCapitulos()) {  // inserido

      String html = capitulo.getConteudoHTML(); // inserido
```

```
      List<IElement> convertToElements =
          HtmlConverter.convertToElements(html);
      for (IElement element : convertToElements) {
        pdfDocument.add((IBlockElement) element);
      }
      // TODO: não adicionar página depois do último capítulo
      pdfDocument.add(new AreaBreak(AreaBreakType.NEXT_PAGE));

    }  // inserido

  //restante do código...
```

Essa classe deve voltar a ser compilada com sucesso!

No `GeradorEPUB`, devemos criar uma variável para o `arquivoDeSaida` do `Ebook` e percorrer os capítulos, usando o HTML na geração do EPUB, removendo o *TODO*.

```
// cotuba.GeradorEPUB

public class GeradorEPUB {

  public void gera(Ebook ebook) {

    Path arquivoDeSaida = ebook.getArquivoDeSaida(); // inserido

    var epub = new Book();

    for (Capitulo capitulo : ebook.getCapitulos()) { // inserido

      String html = capitulo.getConteudoHTML(); // inserido

      String tituloDoCapitulo = capitulo.getTitulo(); // inserido

      //~~TODO: usar título do capítulo~~
      epub.addSection(tituloDoCapitulo, //modificado
          new Resource(html.getBytes(), MediatypeService.XHTML));

    } // inserido
```

```
    var epubWriter = new EpubWriter();

    //restante do código...
```

Ufa! A classe compilará sem erros. Finalmente!

O PDF está sendo gerado com uma página em branco depois do último capítulo. Há inclusive um TODO referente a isso na classe `GeradorPDF`. Para resolver, devemos definir um método `isUltimoCapitulo` em `Ebook`:

```
// cotuba.Ebook

public class Ebook {
  //...

  public boolean isUltimoCapitulo(Capitulo capitulo) {
    return this.capitulos.get(this.capitulos.size() - 1)
        .equals(capitulo);
  }
}
```

> Talvez seja uma maneira ingênua de manipular os parâmetros. Poderíamos usar uma programação mais defensiva, evitando exceções indesejadas. Fica como exercício!

Em `GeradorPDF`, não precisamos mais adicionar a quebra de página no último capítulo:

```
// cotuba.GeradorPDF

//-TODO:-não-adicionar-página-depois-do-último-capítulo-
if (!ebook.isUltimoCapitulo(capitulo)) { // inserido

  pdfDocument.add(new AreaBreak(AreaBreakType.NEXT_PAGE));

} // inserido
```

Ao testarmos a geração de PDF e EPUB, tudo deve funcionar!

Há uma classe anônima em `RenderizadorMDParaHTML`. Qual é a responsabilidade dela? Vale a pena movê-la para uma classe própria?

A classe anônima de `RenderizadorMDParaHTML` implementa `org.commonmark.node.AbstractVisitor`, que é responsável por descobrir os headings (no Markdown `#`, a `######` e no HTML, `<h1>` a `<h6>`). Está sendo usada para descobrir o título (`#` / `<h1>`) do capítulo que está sendo renderizado. Se decidíssemos mover para uma classe própria, seria possível fazer isso de maneira menos trabalhosa usando as refatorações de uma IDE. Fica como mais um exercício extra!

Para saber mais: Design Simples

Na primeira edição do livro *Extreme Programming Explained* (BECK, 2000), Kent Beck descreve como prática o Design Simples, em que o código deve seguir, na ordem, os seguintes critérios:

1. Passa em todos os testes automatizados
2. Não contém nenhuma duplicação
3. Expressa as ideias com clareza
4. Minimiza o número de classes e métodos

Há o foco em remover a duplicação de código, como fizemos anteriormente, além de termos código expressivo e sem classes desnecessárias.

É uma maneira subjetiva, porém interessante, de guiar o design do código em direção a uma boa distribuição de responsabilidades e à alta coesão.

3.6 MVC, entidades e casos de uso

O MVC (Model-View-Controller) é um *pattern* arquitetural usado como um molde pra distribuição de responsabilidades em trechos de código que tratam de interfaces com o usuário (UI). Há três responsabilidades preestabelecidas:

- *View*: contém a lógica que monta a UI (telas ou equivalente) e que trata a entrada de dados, recebendo eventos do usuário (clique, digitação etc.). As interações do usuário são repassadas para o *Controller*. A *View* pode buscar dados diretamente do *Model* para exibição na UI.
- *Controller*: o "meio de campo", recebe interações do usuário da *View* e colabora com o *Model* para enviar e obter dados, que são repassados para a *View*.
- *Model*: o resto do código, que não tem a ver com UI. Realiza cálculos, regras de negócio, persistência, integrações com outros sistemas etc.

Há diversas variações como MVP, MVVM, entre outros.

No livro *Patterns of Enterprise Application Architecture* (FOWLER et al., 2002), Martin Fowler destaca que a **separação da apresentação do modelo** é uma das ideias mais fundamentais do bom design porque:

- apresentação e modelo têm a ver com diferentes preocupações
- o mesmo modelo pode ter diferentes apresentações
- é mais fácil de criar testes automatizados para a lógica de domínio do que para o código de UI

Apesar de MVC ser um molde de distribuição de responsabilidades muito comum, muitas vezes não é trivial aplicá-lo. Maurício Aniche, no livro *OO e SOLID para Ninjas* (ANICHE, 2015), diz que um dos

problemas de coesão mais comuns são **Controllers que fazem coisas demais**:

- buscam dados do BD
- implementam regras de negócio
- enviam emails
- chamam Web Services
- enviam resultados para a View (o que realmente deveriam fazer).

MVC para linha de comando?

É uma boa prática separar a lógica de negócio da apresentação, pensando em distribuição de responsabilidades. Mas em uma aplicação de linha de comando como o Cotuba, será que isso é mesmo necessário?

O que seria a UI, nesse caso? E a entrada de dados do usuário? A UI não é composta por telas, mas pelas mensagens que imprimimos para o usuário. A entrada de dados é feita por meio das opções passadas ao executar o comando.

MVC no Cotuba

A UI do Cotuba são os `System.out.println()` e `System.err.println()`. A entrada de dados são os parâmetros que recebemos no `String[] args`, que repassamos para `LeitorOpcoesCLI`, que os analisa.

Como a UI é muito simples, nossa View seriam esses trechos de código da `Main` que recebem os parâmetros do usuário e imprimem mensagens. O nosso Model seriam as classes responsáveis por representar os ebooks, capítulos, renderização de Markdown, geração de PDFs e de EPUBs. Quem seria nosso Controller? Seria a classe responsável pela coordenação, obtendo os parâmetros recebidos do usuário, interagindo com os objetos do Model e imprimindo mensagens: a `Main` também!

Entidades e casos de uso

Uncle Bob, no livro *Clean Architecture* (MARTIN, 2017), cita o trabalho de Ivar Jacobson para argumentar que regras de negócio são compostas por dois tipos de objeto: as entidades e os casos de uso.

As entidades são objetos que contém dados e funções que disponibilizam regras de negócio críticas. Devem ser totalmente independentes de como os dados são armazenados em um Bancos de Dados, de como são apresentados em uma UI e dos frameworks utilizados.

Já os casos de uso são objetos que especificam as requisições do usuário, as respostas que serão produzidas e os passos de processamento envolvidos em produzir essas respostas. Casos de uso são específicos para uma aplicação e controlam como e quando as regras de negócio das entidades são invocadas.

Casos de uso não descrevem como o sistema vai ser apresentado para o usuário nem como os dados chegam ou saem do sistema. São focados na interação com as entidades.

> *"Casos de uso controlam a dança das entidades."*
>
> Uncle Bob, no livro *Clean Architecture* (MARTIN, 2017)

Entidades e casos de uso no Cotuba

As entidades do Cotuba são as classes `Ebook` e `Capitulo`. Mas há algum caso de uso? Ainda não... Nosso Controller, a classe `Main`, está responsável por extrair os parâmetros e apresentar mensagens para o usuário. Até aí tudo bem... Mas, além disso, a `Main` está responsável por coordenar a renderização dos arquivos Markdown em HTML e a geração dos ebooks.

Um design mais interessante é extrair essa coordenação da `Main` para uma outra classe. Essa nova classe representaria o caso de uso de gerar ebooks a partir dos parâmetros do usuário.

> No caso do Cotuba, só temos um caso de uso. Para sistemas mais complexos, teríamos diversos casos de uso. Em um sistema para uma editora de livros, teríamos casos de uso como cadastro de autores e publicação de livros, por exemplo.

Que nome daríamos para essa classe coordenadora? Podemos dizer que ela representa a própria aplicação. Poderíamos chamá-la com um nome sugestivo: `Cotuba` . Pensando nos moldes do MVC, onde a classe `Cotuba` seria encaixada? Seria parte do Model, já que não faz parte de código relacionado a UI, mas de todo o resto do sistema!

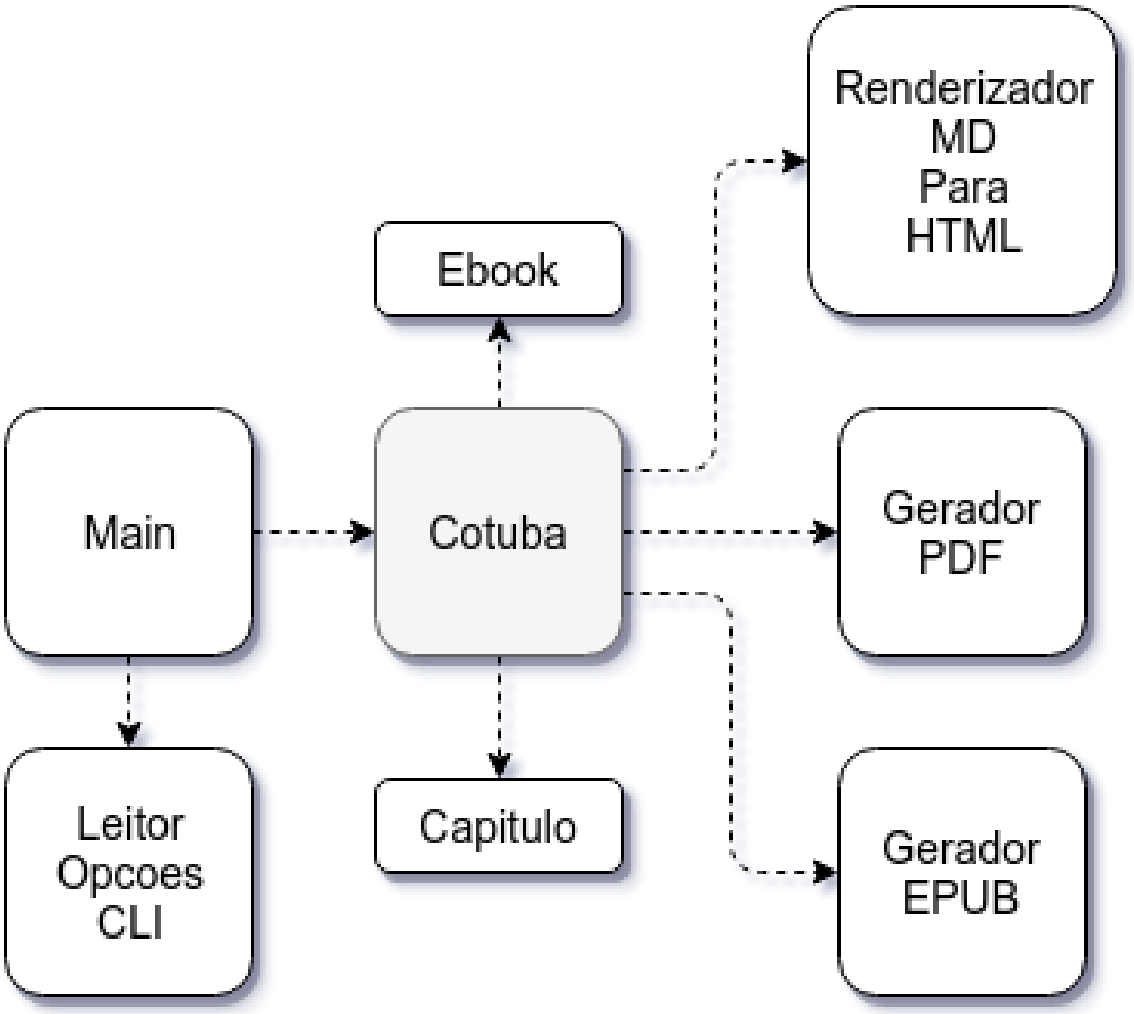

Figura 3.4: Cotuba coordena a geração do ebook

> *"Ciência da Computação é a disciplina que acredita que todos os problemas podem ser resolvidos com mais uma camada de indireção."*
>
> Dennis DeBruler, citado por Kent Beck no livro *Refactoring* (FOWLER et al., 1999)

Separando a Aplicação do Controller no Cotuba

Vamos implementar a nova classe `Cotuba`, que será responsável pela coordenação da geração de ebooks a partir dos parâmetros. Para isso, devemos criar a classe `Cotuba`, no pacote `cotuba`, definindo um método `executa` que recebe os parâmetros da linha de comando:

```
// cotuba.Cotuba

package cotuba;

import java.nio.file.Path;

public class Cotuba {

  public void executa(String formato, Path diretorioDosMD,
      Path arquivoDeSaida) {

  }

}
```

Então, vamos mover o código a seguir de `Main` para o método `executa` de `Cotuba`:

```
// cotuba.Cotuba

var renderizador = new RenderizadorMDParaHTML();
List<Capitulo> capitulos =
    renderizador.renderiza(diretorioDosMD);

Ebook ebook = new Ebook();
ebook.setFormato(formato);
ebook.setArquivoDeSaida(arquivoDeSaida);
ebook.setCapitulos(capitulos);

if ("pdf".equals(formato)) {

  var geradorPDF = new GeradorPDF();
  geradorPDF.gera(ebook);
```

```
} else if ("epub".equals(formato)) {

  var geradorEPUB = new GeradorEPUB();
  geradorEPUB.gera(ebook);

} else {

  throw new IllegalArgumentException(
    "Formato do ebook inválido: " + formato);

}
```

Detalhe: o `try` deve ser mantido na `Main`. Em `Main`, devemos chamar a nova classe `Cotuba`. O nosso Controller, a classe `Main`, deverá ficar assim:

```
// cotuba.Main

public class Main {

  public static void main(String[] args) {

    Path diretorioDosMD;
    String formato;
    Path arquivoDeSaida;
    boolean modoVerboso = false;

    try {

      var opcoesCLI = new LeitorOpcoesCLI(args);

      Path diretorioDosMD = opcoesCLI.getDiretorioDosMD();
      String formato = opcoesCLI.getFormato();
      Path arquivoDeSaida = opcoesCLI.getArquivoDeSaida();
      boolean modoVerboso = opcoesCLI.isModoVerboso();

      // modificado
      var cotuba = new Cotuba();
      cotuba.executa(formato, diretorioDosMD, arquivoDeSaida);

      System.out.println("Arquivo gerado com sucesso: "
```

```
            + arquivoDeSaida);

      } catch (Exception ex) {
        System.err.println(ex.getMessage());
        if (modoVerboso) {
          ex.printStackTrace();
        }
        System.exit(1);
      }
    }

}
```

Agora já podemos testar a geração de PDF e EPUB. Deve funcionar! Compare a `Main` com a nossa versão inicial. As responsabilidades estão muito mais definidas, não é mesmo?

Melhorando um detalhe na impressão de mensagens

Há um pequeno detalhe na classe `LeitorOpcoesCLI` que quebra nosso design: quando o usuário passa uma opção inválida, por exemplo, acontece uma `ParseException`, imprimimos uma mensagem e matamos o programa. Será que isso não é responsabilidade do nosso Controller, a classe `Main`?

Vamos trocar esse código por uma `IllegalArgumentException`, mantendo a impressão da ajuda:

```
// cotuba.LeitorOpcoesCLI

try {
  cmd = cmdParser.parse(options, args);
} catch (ParseException e) {
  System.err.println(e.getMessage());

  ajuda.printHelp("cotuba", options);

  System.exit(1);
  return;
```

```
  throw new IllegalArgumentException("Opção inválida", e);
}
```

Tudo deve continuar funcionando da mesma maneira após essa modificação!

Para saber mais: GRASP

Definirmos algo semelhante a um Controller, como fizemos anteriormente, é uma ideia recorrente no design de código Orientado a Objetos.

Craig Larman, no livro *Applying UML and Patterns* (LARMAN, 2004), documenta soluções recorrentes para atribuição de responsabilidades a objetos: são os *General Responsibility Assignment Software Patterns/Principles* (GRASP). Podemos agrupá-los em princípios mais gerais e *patterns* mais específicos.

Princípios gerais

- *Alta coesão*
- *Baixo acoplamento*
- *Polimorfismo*
- *Indireção*: um intermediário que media as interações entre outros objetos, evitando acoplamento direto.
- *Variações protegidas*: uma interface estável em volta de pontos de variação do design.

Patterns específicos

- *Information Expert*: uma classe que tem a maior quantidade de informação para cumprir uma responsabilidade.
- *Controller*: um "ponto de entrada" do sistema que recebe "eventos" do usuário ou uma representação de um "caso de uso" no código.
- *Creator*: encapsula a criação de outros objetos.

- *Pure fabrication*: classe que não existe no domínio de negócio mas promove baixo acoplamento e alta coesão.

Larman parte de uma abordagem criada por Rebecca Wirfs-Brock chamada de *Responsibility Driven Design* (RDD):

> *"RDD é uma metáfora geral para pensar sobre Design Orientado a Objetos (OOD). Pense em objetos como pessoas com responsabilidades que colaboram com outras pessoas para realizar suas tarefas. RDD enxerga OOD como uma comunidade de objetos colaborativos e responsáveis."*
>
> Craig Larman, no livro *Applying UML and Patterns* (LARMAN, 2004)

3.7 Responsabilidades em pacotes e métodos

Até agora, tudo está no mesmo pacote. Uma maneira mais interessante de organizar as classes é ter pacotes com responsabilidades bem definidas. Seria como o SRP um nível acima, nos artefatos que contém classes: os pacotes. Vamos mover as classes para que fiquem nos seguintes pacotes:

```
cotuba
├── application
│   └── Cotuba.java
├── cli
│   ├── LeitorOpcoesCLI.java
│   └── Main.java
├── domain
│   ├── Capitulo.java
│   └── Ebook.java
├── epub
│   └── GeradorEPUB.java
├── md
│   └── RenderizadorMDParaHTML.java
└── pdf
    └── GeradorPDF.java
```

Não podemos deixar de mudar o conteúdo do arquivo `src/scripts/cotuba.sh` para que tenha o pacote correto da classe `Main`:

```
# src/scripts/cotuba.sh

#!/bin/bash
java -cp "libs/*" cotuba.cli.Main "$@"
```

Tudo público?

Será que a classe `LeitorOpcoesCLI` precisa mesmo ser `public`?

No artigo *Modularity and testability* (BROWN, 2014), Simon Brown faz uma provocação: *"devemos fazer uma doação para a caridade toda vez que digitarmos* `public class` *sem pensar se essa classe realmente precisa ser pública."*

No Java, o modificador de acesso padrão torna a classe acessível apenas por classes do mesmo pacote. Por isso, é chamado de *package private*. Com menos código acessível, as dependências entre as classes tendem a ser minimizadas, levando a um código mais organizado e, por isso, mais fácil de entender e modificar. Isso favorece o encapsulamento no nível de pacotes!

Já que a classe `LeitorOpcoesCLI` só é usada dentro do pacote `cotuba.cli`, podemos remover o modificador de acesso `public`, mantendo-a acessível apenas dentro do código do próprio pacote.

```
// cotuba.cli.LeitorOpcoesCLI

public class LeitorOpcoesCLI {

  // restante do código ...

}
```

Responsabilidades no nível de métodos

Conforme mencionado anteriormente, o lema "fazer uma coisa só" não é exatamente o intuito do SRP. O SRP é focado no nível de classes e, mais especificamente, em ter poucos motivos para mudá-las.

Porém, para Uncle Bob, um método deveria fazer apenas uma coisa. Se estiver fazendo muitas coisas, devemos quebrá-los em métodos auxiliares, menores e, provavelmente, privados.

> Não vamos detalhar as refatorações a seguir. Fica a cargo de quem está lendo!

No construtor de `LeitorOpcoesCLI`, são criadas as opções possíveis e, depois, extraídas do `String[] args` os valores dessas opções. Poderíamos quebrar esse código em métodos auxiliares. Nesta classe, extrairíamos um método para criar as opções e outros métodos para tratá-las, invocando-os no construtor:

```
// cotuba.cli.LeitorOpcoesCLI

public class LeitorOpcoesCLI {

  // atributos omitidos...

  public LeitorOpcoesCLI(String[] args) {
    Options options = criaOpcoes();
    CommandLine cmd = parseDosArgumentos(args, options);
    trataDiretorioDosMD(cmd);
    trataFormato(cmd);
    trataArquivoDeSaida(cmd);
    trataModoVerboso(cmd);
  }

  private Options criaOpcoes() {
    // instancia Options e adiciona cada uma das opções
  }
  private CommandLine parseDosArgumentos(String[] args,
      Options options) {
    // faz parse dos args e os transforma em um CommandLine
  }
  private void trataDiretoriosDosMD(CommandLine cmd) {
    // trata a opção 'dir' e define o atributo 'diretorioDosMD'
  }
  private void trataFormato(CommandLine cmd) {
    // trata a opção 'format' e define o atributo 'formato'
  }
  private void trataArquivoDeSaida(CommandLine cmd) {
    // trata a opção 'output' e define atributo 'arquivoDeSaida'
  }
  private void trataModoVerboso(CommandLine cmd) {
    // trata a opção 'verbose' e define o atributo 'modoVerboso'
```

```
  }
}
```

Já no método `renderiza` de `RenderizadorMDParaHTML`, são buscados os arquivos `.md` do diretório e, depois, é invocado o *parse* e a renderização para HTML de cada `.md`. Esse código poderia ser quebrado em métodos auxiliares. Nesta classe, definiríamos um método para obter os arquivos Markdown, um para realizar o *parsing* e outro para renderizar o HTML:

```
// cotuba.md.RenderizadorMDParaHTML

public class RenderizadorMDParaHTML {

  public List<Capitulo> renderiza(Path diretorioDosMD) {
    return obtemArquivosMD(diretorioDosMD).stream()
        .map(arquivoMD -> {
          Capitulo capitulo = new Capitulo();
          Node document = parseDoMD(arquivoMD, capitulo);
          renderizaParaHTML(arquivoMD, capitulo, document);
          return capitulo;
        }).toList();
  }

  private List<Path> obtemArquivosMD(Path diretorioDosMD) {
    //faz o PathMatcher e retorna uma List<Path> com os arquivos
ordenados...
  }
  private Node parseDoMD(Path arquivoMD, Capitulo capitulo) {
    //usa o Parser e retorna o Node que representa o documento
  }
  private void renderizaParaHTML(Path arquivoMD,
      Capitulo capitulo, Node document) {
    //usa o HtmlRenderer e seta o conteudo HTML no Capitulo
  }
}
```

A geração de PDFs e EPUBs deve continuar funcionando!

3.8 Contraponto: críticas ao SRP

Em seu vídeo *SOLID #1: Uma reflexão sobre o Princípio da responsabilidade única* (SOUZA, 2020), Alberto Souza critica a subjetividade do SRP. Aponta ainda a confusão na interpretação do princípio pela indústria de software, o que muitas vezes leva a uma proliferação de classes com apenas um método. Alberto indica o uso de abordagens mais como a técnica CDD (*Cognitive Driven Development*), que sugere um design de código focado em torno da facilidade de entendimento do código. Uma decomposição do código focada no entendimento levaria a uma melhor distribuição das responsabilidades, aumentando a possibilidade de classes autocontidas e minimizando o efeito colateral de mudanças em outras classes.

No livro *SOLID is not solid* (COPELAND, 2019), David Copeland também critica o SRP por ser vago na definição do que é "responsabilidade" e no que seriam as razões para modificar uma classe. Copeland indica que o SRP pode levar a um monte de pequenas classes que não representam um conceito completo e, por isso, são pouco coesas. Para Copeland, antes de mudar algum código, deveríamos refletir sobre se um conceito completo é implementado e sobre como a mudança afeta a coesão. Copeland ainda afirma que a coesão do código deve ser organizada ao redor de conceitos do negócio.

No artigo *CUPID - the back story* (NORTH, 2021), Dan North chama o SRP de Princípio Inutilmente Vago. O foco em ter apenas uma responsabilidade seria difícil de implementar: um processador de dados no estilo ETL (Extract, Transform, Load) seria uma ou três responsabilidades? O foco em razões para mudar não faria sentido, porque qualquer código não trivial teria inúmeras razões para ser modificado. North sugere um princípio alternativo: escreva código

simples de entender, dado que haja conhecimento na linguagem de programação e no domínio de negócio. Além disso, o código deveria manter integridade conceitual, não sendo decomposto de maneiras arbitrárias.

3.9 O que aprendemos?

Pronto! O código do Cotuba está razoavelmente aderente ao SRP depois de nossas refatorações. Dessa forma, o código é composto por classes menores e com responsabilidades mais bem distribuídas. Mudanças nas necessidades do projeto estarão restritas a alterações em poucas classes.

Mas e os outros princípios do SOLID? No próximo capítulo, vamos do "S" ao "D" para estudar o Princípio da Inversão de Dependências.

O código deste capítulo pode ser encontrado em: https://github.com/alexandreaquiles/solid-na-pratica/tree/cap3-single-responsibility-principle

Figura 3.5: Vídeo do capítulo

Capítulo 4

Princípio da Inversão de Dependências (DIP): dependências estáveis

Considere que temos a seguinte classe em um projeto interno da nossa empresa:

```
public class EmissorNotaFiscal {

  private RegrasDeTributacao tributacao;
  private LegislacaoFiscal legislacao;
  private NotaFiscalDAO notas;
  private EnviadorEmail email;
  private EnviadorSMS sms;

  //...

  public NotaFiscal gera(Fatura fatura) {

    List<Imposto> impostos = tributacao.verifica(fatura);
    List<Isencao> isencoes = legislacao.analisa(fatura);

    // método auxiliar
    NotaFiscal nota = aplica(impostos, isencoes);

    notas.salva(nota);

    String mensagemNota = String.format(
          "Sua nota fiscal foi gerada: %s",  nota.getNumero());

    net.sargue.mailgun.Configuration configuration =
      new net.sargue.mailgun.Configuration()
        .domain("empresa.com")
        .apiKey(System.getenv("MAILGUN_API_KEY"))
        .from("Empresa", "contato@empresa.com");

    net.sargue.mailgun.Mail mail =
      net.sargue.mailgun.Mail.using(configuration)
        .to(fatura.getEmail())
        .subject("Sua nota fiscal")
        .text(mensagemNota)
```

```
            .build();

        email.envia(mail);

        com.twilio.rest.api.v2010.account.Message smsMessage =
          com.twilio.rest.api.v2010.account.Message.creator(
                new PhoneNumber("+14159352345"),
                new PhoneNumber(fatura.getTelefone()),
                mensagemNota)
            .create();

        sms.envia(smsMessage);

        return nota;
    }

}
```

> Esse exemplo é inspirado no livro *OO e SOLID para Ninjas* (ANICHE, 2015).

Perceba que a classe `EmissorNotaFiscal` tem como dependências as classes `RegrasDeTributacao`, `LegislacaoFiscal`, `NotaFiscalDAO`, `EnviadorEmail` e `EnviadorSMS`.

Cada uma dessas classes aparenta ter uma responsabilidade bem definida, só um motivo para serem modificadas e, portanto, parecem seguir o SRP. Mas será que o design do código está bom o bastante? Há um problema claro: a classe `EmissorNotaFiscal` tem **muitas dependências**.

4.1 Acoplamento, estabilidade e volatilidade

Uma classe com muitas dependências tem **acoplamento** com muitas outras classes. Além disso, o código acaba acoplado também às dependências das dependências e assim por diante. A classe `EmissorNotaFiscal` depende indiretamente do Hibernate, por meio de `NotaFiscalDAO`; da API do Mailgun, por meio de `EnviadorEmail`; e da API REST do Twilio, por meio de `EnviadorSMS`.

Mudanças nas dependências, ou nas dependências das dependências, podem acabar se propagando para a classe que as utiliza. Precisamos ter atenção para evitar que dependências indesejadas se espalhem indevidamente pelo nosso código. Perceba que essa propagação de dependências acontece em dois pontos na classe `EmissorNotaFiscal`:

- criamos instâncias das classes `Configuration` e `Mail`, ambas do pacote `net.sargue.mailgun` da API do Mailgun, passando o `Mail` para o método `envia` de `EnviadorEmail`.
- criamos uma instância da classe `Message`, do pacote `com.twilio.rest.api.v2010.account` da SDK Java do Twilio, para passá-la para o método `envia` do `EnviadorSMS`.

Essas dependências acabam vazando de `EnviadorEmail` e `EnviadorSMS`, respectivamente. No fim das contas, são quebras do SRP: mudanças na criação de `Mail` e de `Message` não deveriam ser motivos para modificar a classe `EmissorNotaFiscal`, cujas mudanças deveriam estar relacionadas ao fluxo de geração de notas fiscais.

O efeito desse vazamento de dependências aparece quando precisamos adaptar nosso código para atender a novas necessidades técnicas. Se quiséssemos, por exemplo, trocar a API de envio de SMS para Vonage ou Amazon SNS, teríamos, claro, que modificar o `EnviadorSMS`.

Até aí tudo bem... Mas a necessidade de mudança também seria espalhada para a classe `EmissorNotaFiscal`, o que seria inesperado! Teríamos uma situação parecida em relação a mudanças no envio de emails. A falta de cuidados com as dependências do nosso código faz com que a flexibilidade e o reúso fiquem prejudicados.

Acoplamento bom x acoplamento ruim

Acoplamento precisa existir. Uma classe totalmente desacoplada é uma classe inútil.

Só existe uma maneira de evitarmos totalmente o acoplamento: colocar todo o código incluindo o das bibliotecas que usamos, em uma mesma classe. Se tudo estiver junto, não há a necessidade de depender de nada externo. Mas isso levaria a uma quebra do SRP, a uma baixíssima coesão e a um pesadelo de manutenção!

E será que todo acoplamento é ruim? Não! Em um código Java, desde o primeiro `OlaMundo`, dependemos de `String` e de `System`. Esse acoplamento não chega a ser problemático, não é mesmo? As classes do pacote `java.lang` são **estáveis**: mudam muito pouco.

Perceba que, quando falamos de classes, estabilidade não é uma característica boa ou ruim. Não há aqui nenhum juízo de valor. Classes estáveis têm essa característica porque milhões de projetos as usam e mudá-las teria um impacto gigantesco. Por isso, podemos depender delas tranquilamente. É o *acoplamento bom*.

Vamos comparar classes como `String` ou `System` com as dependências da classe `EmissorNotaFiscal`. Qual é a chance, por exemplo, de as classes `EnviadorEmail` ou `EnviadorSMS` mudarem? É grande! Essas classes são **voláteis**. Depender de classes voláteis é o *acoplamento ruim*.

VOLÁTIL

FIG Pouco firme; inconstante, mudável, volúvel.

Fonte: Michaelis Online

4.2 Abstrações e inversão das dependências

Como minimizar os impactos de mudanças em dependências voláteis? Usando **abstrações**! Podemos usar classes abstratas e, preferencialmente, interfaces. Abstrações são estáveis: mudam muito menos que implementações.

> Podemos também usar classes concretas como abstrações, desde que não deixemos detalhes de outras dependências vazarem.

Uma abstração para o envio de SMS

Voltando para o nosso exemplo anterior, poderíamos criar uma abstração para o envio de SMS que precisa ser realizado pela classe `EmissorNotaFiscal`.

Como chamar a interface que abstrai o envio de SMS? Uma maneira comum em projetos .NET é colocar um `I` como prefixo. Dessa forma, teríamos `IEnviadorSMS`. Manteríamos a implementação com um nome simples como `EnviadorSMS`.

Uma outra maneira, mais comum em projetos Java, é chamar a abstração de um nome simples como `EnviadorSMS`. Já a implementação teria o sufixo `Impl`. A implementação seria nomeada como `EnviadorSMSImpl`. Será nossa abordagem!

> Depois discutiremos se o uso de `Impl` é uma boa prática ou não.

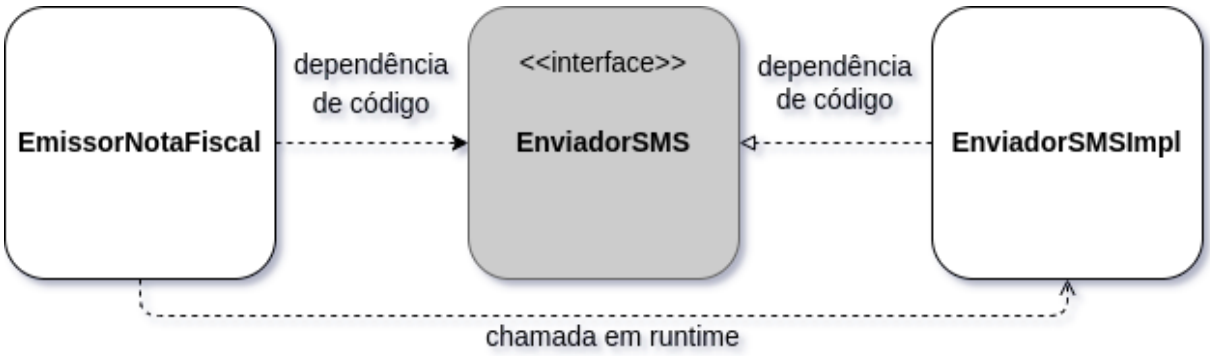

Figura 4.1: Dependência de código invertido x runtime.

Repare bem nas setas da imagem anterior. Ambas as dependências de código apontam na direção da abstração: a interface `EnviadorSMS`.

Ao usarmos classes abstratas ou interfaces, o código não depende mais diretamente da dependência volátil e sim da abstração. E a dependência volátil, por sua vez, também depende da abstração, implementando-a. Por isso, podemos dizer que a **dependência é invertida**.

Em *runtime*, as chamadas ainda são realizadas em uma implementação concreta. Veremos como ligar uma implementação à sua abstração mais adiante.

Evitando vazamento de detalhes de implementação

Uma coisa importante é que devemos evitar que detalhes das dependências vazem nas nossas abstrações. No caso do envio de SMS, a interface `EnviadorSMS` não deveria receber uma `Message` da SDK Java do Twilio. Em vez disso, deveríamos receber código mais próximo das regras de negócio, como `mensagem` e `telefoneDestinatario`:

```
import com.twilio.rest.api.v2010.account.Message;

public interface EnviadorSMS {

  void envia(Message message);
```

```
  void envia(String mensagem, String telefoneDestinatario);

}
```

A criação da `Message` do Twilio deve ser um detalhe de implementação da classe `EnviadorSMSImpl`. Não deve ser possível saber, a partir da assinatura dos métodos da interface `EnviadorSMS`, se é usado Twilio, Vonage ou Amazon SNS.

O mesmo vale para o envio de e-mails feito pela classe `EmissorNotaFiscal`. Criaríamos uma interface `EnviadorEmail`, tomando os devidos cuidados para que detalhes de implementação, como o uso da API do Mailgun, não sejam expostos indevidamente.

```
import net.sargue.mailgun.Mail;

public interface EnviadorEmail {

  void envia(Mail mail);
  void envia(String mensagem, String emailDestinatario);

}
```

Ao usarmos as novas interfaces `EnviadorEmail` e `EnviadorSMS`, nosso exemplo do início do capítulo, a classe `EmissorNotaFiscal`, ficaria parecida com o seguinte código:

```
public class EmissorNotaFiscal {

  private RegrasDeTributacao tributacao;
  private LegislacaoFiscal legislacao;
  private NotaFiscalDAO notas;
  private EnviadorEmail email; // agora é uma interface
  private EnviadorSMS sms; // agora é uma interface

  //...

  public NotaFiscal gera(Fatura fatura) {

    List<Imposto> impostos = tributacao.verifica(fatura);
```

```
    List<Isencao> isencoes = legislacao.analisa(fatura);

    // método auxiliar
    NotaFiscal nota = aplica(impostos, isencoes);

    notas.salva(nota);

    String mensagemNota = String.format(
          "Sua nota fiscal foi gerada: %s",  nota.getNumero());

    // modificado
    email.envia(mensagemNota, fatura.getEmail());
    sms.envia(mensagemNota, fatura.getTelefone());

    return nota;
  }

}
```

Abstrações conceitualmente abstratas

Um detalhe mais sutil na criação de abstrações é que precisamos evitar vinculá-las conceitualmente a tarefas muito particulares. Poderíamos, por exemplo, ter definido a interface `EnviadorSMS` usando termos mais relacionados a notas fiscais:

```
public interface EnviadorSMS {
  void enviaNota(String mensagemNota,
      String telefoneClienteNota); // abstração ruim
}
```

O contrato definido pela interface é muito semelhante ao anterior, mas os termos são menos abrangentes. E se quiséssemos enviar um SMS para token de autenticação em duas etapas? Deveríamos definir outro método específico, como `enviaTokenAutenticacao` ? Não parece adequado, se o envio de SMS para notas fiscais e tokens de autenticação forem exatamente iguais. Isso é um indicador de uma abstração ruim.

Boas abstrações devem fornecer capacidades mais amplas, sem estarem atreladas a um uso muito específico.

Abstrações, uma ideia antiga

A ideia de criar programas ao redor de abstrações vem de longa data em artigos e livros da comunidade de Orientação a Objetos. No artigo *Design Principles and Design Patterns* (MARTIN, 2000), Uncle Bob declara que devemos *"Depender de abstrações e não de implementações."* No livro clássico *Design Patterns* (GAMMA et al., 1994), os autores escrevem algo semelhante em um dos capítulos iniciais: *"Programe voltado à interface, não à implementação."*

O uso de abstrações não está restrito a linguagens OO. Em outro livro clássico, o *SICP* (ABELSON; SUSSMAN; SUSSMAN, 1996), é usada a linguagem funcional Scheme, um sabor de Lisp criado por um dos autores. Nesse livro, as abstrações são criadas por meio de funções que manipulam estruturas de dados padrão da linguagem, chamadas de *procedimentos de interface*, e não por interfaces ou classes abstratas.

Os autores deixam claro o poder de criar o que chamam de *barreiras de abstração* no código: *"Restringir a dependência da representação a alguns procedimentos de interface nos ajuda a projetar programas, bem como a modificá-los, porque nos permite manter a flexibilidade para considerar implementações alternativas."*

4.3 Regras de negócio x detalhes

Quando desenvolvemos uma aplicação, a parte mais importante do nosso código é a que implementa as **regras de negócio**. Porém, grande parcela do código não está relacionada ao negócio, mas a coisas mais técnicas, **detalhes de implementação** como UI Web e/ou Mobile, persistência e Banco de Dados, integração com outros sistemas, frameworks, protocolos etc.

Por isso, em suas palestras, Uncle Bob costuma dizer: a Web é um detalhe; o Banco de Dados é um detalhe. Deveríamos preparar o nosso código de negócio para "sobreviver" a mudanças completas nesses detalhes.

Mas sem esses detalhes, não conseguiríamos expor uma UI ou persistir os dados. Os detalhes técnicos são importantes, são os **mecanismos de entrega** das regras de negócio para os usuários.

4.4 Código de alto nível e baixo nível

Uncle Bob, no livro *Agile Principles, Patterns, and Practices in C#* (MARTIN, 2006), divide o código em dois tipos:

- Códigos de **alto nível** seriam os códigos que implementam regras de negócio.
- Códigos de **baixo nível** seriam os mecanismos de entrega, os detalhes de implementação mais técnicos.

Vamos voltar ao nosso exemplo de emissão de notas fiscais. Das dependências da classe `EmissorNotaFiscal`, poderíamos classificar como de alto nível, relacionadas ao negócio: `RegrasDeTributacao` e `LegislacaoFiscal`.

Já as dependências de baixo nível de `EmissorNotaFiscal` seriam: `NotaFiscalDAO`, `EnviadorEmail` e `EnviadorSMS`.

Alto/baixo nível e entradas/saídas

No livro *Clean Architecture* (MARTIN, 2017), Uncle Bob diz que código de alto nível é aquele mais distante das entradas ou saídas do sistema e, por isso, muda menos frequentemente e por razões mais importantes, relacionadas ao negócio.

Já o código de baixo nível, mais próximo das entradas ou saídas, muda mais frequentemente e com mais urgência.

Alto ou baixo nível depende do contexto

Para aplicações Web, o código SQL pode ser considerado um código de baixo nível, um detalhe de implementação, já que não está diretamente ligado às regras de negócio, mas à representação das entidades em um banco de dados relacional.

Porém, para quem desenvolve o Hibernate, gerar código SQL comum entre os bancos de dados pode ser considerado alto nível, já que está relacionado ao problema que uma biblioteca de ORM resolve. Variações entre os bancos de dados, como `AUTO_INCREMENT`, `SEQUENCE` ou `IDENTITY` na geração de PKs, seriam detalhes de implementação de baixo nível.

4.5 O Princípio da Inversão de Dependências (DIP)

Pensando em uma boa maneira de gerenciar as dependências no nosso código, Uncle Bob definiu o *Dependency Inversion Principle (DIP)*, o Princípio da Inversão de Dependências:

> **DEPENDENCY INVERSION PRINCIPLE (DIP)**
>
> *Módulos de alto nível não devem depender de módulos de baixo nível. Ambos devem depender de abstrações.*
>
> *Abstrações não devem depender de detalhes. Detalhes devem depender de abstrações.*

Se seguirmos o DIP à risca, tanto o código de alto nível como o código de baixo nível deveriam depender de abstrações. Mas, muitas vezes, essa regra é relaxada para simplificar o código. Em outras palavras, regras de negócio não devem depender de mecanismos de entrega, mas de abstrações desses detalhes de implementação.

Em relação às dependências da classe `EmissorNotaFiscal`, para seguir o DIP, deveríamos criar abstrações para as dependências de baixo nível. Para isso, teríamos que definir:

- A interface `NotaFiscalDAO` como abstração e `NotaFiscalDAOImpl` como implementação que depende do Hibernate.
- A interface `EnviadorEmail` como abstração e `EnviadorEmailImpl` como implementação que depende da API do Mailgun.
- A interface `EnviadorSMS` como abstração de `EnviadorSMSImpl` e como implementação que depende da API do Twilio.

> *"Se você está programando uma classe qualquer com regras de negócio e precisa depender de outro módulo, idealmente esse outro módulo deve ser uma abstração."*
>
> Maurício Aniche, no livro *OO e SOLID para Ninjas* (ANICHE, 2015)

Módulos

O que significa o termo "módulo" usado por Uncle Bob na definição do DIP? No livro *Clean Architecture* (MARTIN, 2017), Uncle Bob descreve um módulo como um conjunto coeso de funções e dados ou, de maneira mais simples, um arquivo de código-fonte.

É um conceito que tenta abranger tanto linguagens OO, como linguagens estruturadas e funcionais. Mais adiante veremos um outro conceito para módulos, mais preciso e específico para a linguagem Java.

Toda interface é uma abstração de alto nível?

As APIs da plataforma Java são repletas de interfaces. Pense em `java.sql.Connection` do JDBC ou `javax.jms.Destination` do JMS. São interfaces altamente estáveis: a probabilidade de serem alteradas é bem baixa.

Será que um código de regra de negócio que depende de algumas dessas interfaces segue o DIP? Não! Uma abstração de alto nível é descrita em termos de negócio. Já APIs como o JDBC ou o JMS, mesmo estáveis, são de baixo nível porque são detalhes técnicos, mecanismos de entrega.

DIP, camadas e regra da dependência

Pense em uma arquitetura em três camadas: o código da camada de Apresentação depende do código da camada de Negócio que,

por sua vez, depende do código da camada de Persistência. Essa arquitetura NÃO atende ao DIP: código de alto nível (Negócio) depende de código de baixo nível (Persistência).

Para que atenda ao DIP, teríamos que inserir abstrações na camada de Negócio para inverter as dependências, fazendo com que Persistência dependa de Negócio e não o contrário.

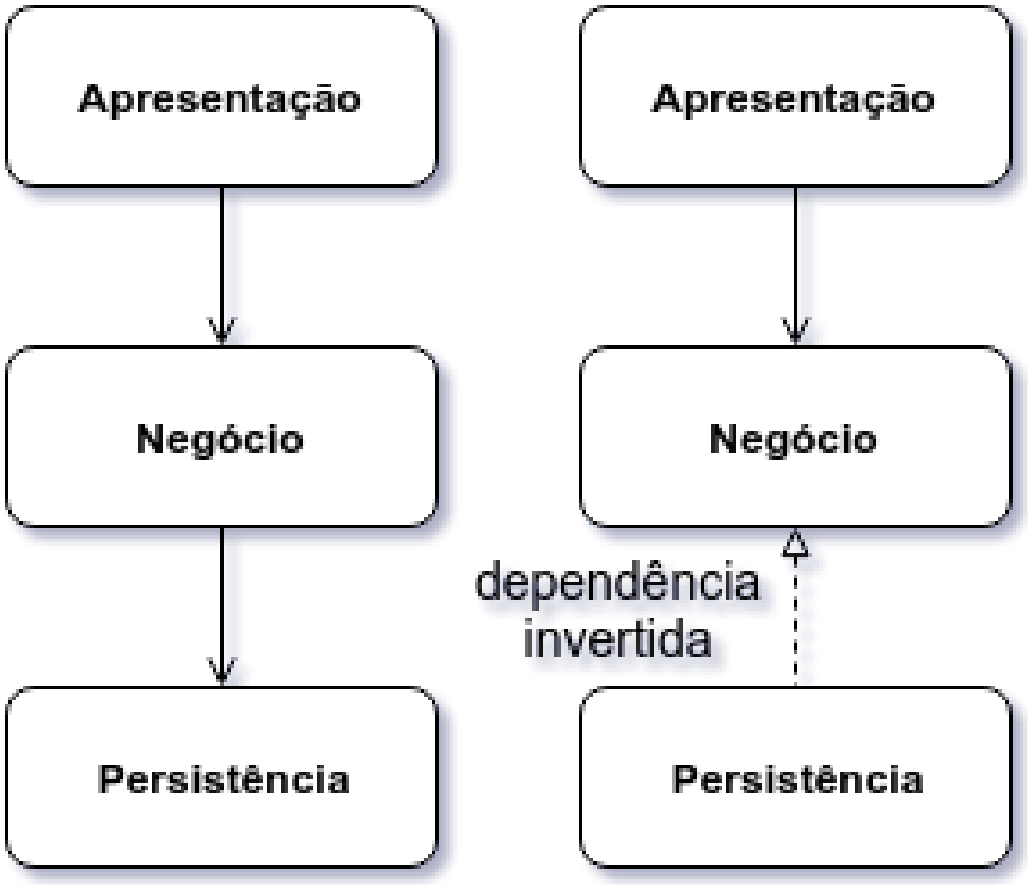

Figura 4.2: Três Camadas x Dependências Invertidas.

Como mencionamos anteriormente, para seguir de maneira estrita o DIP, deveríamos fazer com que a camada de Negócio (alto nível) também fornecesse abstrações para a camada de Apresentação (baixo nível), invertendo também essa dependência. Porém, é comum mantermos uma dependência direta do baixo nível ao alto nível, para que o código fique simplificado.

Regra da dependência

No livro *Clean Architecture* (MARTIN, 2017), Uncle Bob define a **regra da dependência**, que está relacionada ao DIP: *"Dependências devem apontar apenas para dentro, em direção às regras de negócio."*

O termo "para dentro" aqui está relacionado às setas que representam dependências de código em um diagrama. Todas as setas deveriam apontar na direção de código de alto nível, relacionado a regras de negócio.

4.6 DIP no Cotuba

No capítulo anterior, fizemos refatorações que levaram o código do Cotuba em direção ao SRP. Agora vamos analisar a aderência ao DIP. Primeiramente, vamos classificar os níveis do código:

- *Alto nível*: as classes `Cotuba`, `Ebook` e `Capitulo`, que são relativas ao domínio do problema.
- *Baixo nível*: as classes relativas a UI, `Main` e `LeitorOpcoesCLI`, e as classes que implementam detalhes técnicos como `RenderizadorMDParaHTML`, `GeradorPDF` e `GeradorEPUB`.

Será que o código do Cotuba fere o DIP?

A classe `Capitulo` não depende de nada além de `String`, que é uma dependência muito estável. Não a consideramos de baixo nível. Tudo ok.

A classe `Ebook` depende de `List` e `ArrayList`, que são classes bem estáveis e que também não são de baixo nível. Depende também de `Path`, da API NIO, que é uma dependência razoavelmente estável. Vamos considerar como não sendo de baixo nível. Sem problemas.

Porém, a classe `Cotuba`, uma classe de alto nível, depende de várias classes que classificamos como de baixo nível: `RenderizadorMDParaHTML`, `GeradorPDF` e `GeradorEPUB`. Precisamos inverter essas dependências!

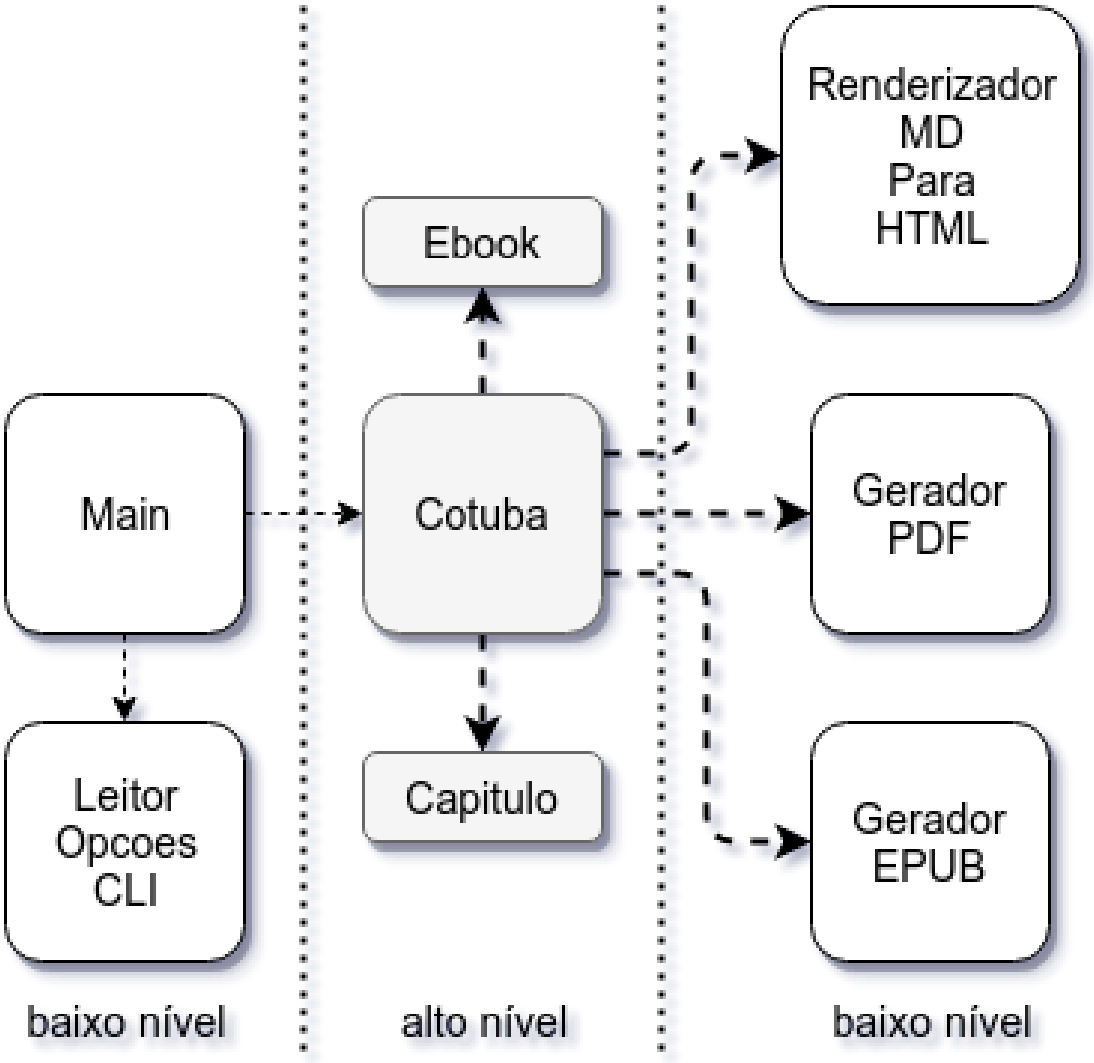

Figura 4.3: Dependências da classe Cotuba.

Invertendo as dependências da classe Cotuba

Vamos iniciar nossa jornada rumo ao DIP, invertendo as dependências da classe `Cotuba`. Para isso, vamos colocar o sufixo `Impl` nas classes das dependências de baixo nível. Em seguida, vamos criar interfaces para essas implementações, fazendo com que `Cotuba` dependa dessas interfaces.

Nosso passo inicial é renomear as dependências de baixo nível de `Cotuba` da seguinte forma:

- de `RenderizadorMDParaHTML` para `RenderizadorMDParaHTMLImpl`
- de `GeradorPDF` para `GeradorPDFImpl`
- de `GeradorEPUB` para `GeradorEPUBImpl`

Depois, vamos extrair as interfaces `RenderizadorMDParaHTML`, `GeradorPDF` e `GeradorEPUB` das respectivas implementações.

> Dica: nossa IDE pode nos ajudar na extração de interfaces!

As classes devem ficar da seguinte maneira, depois dessas refatorações:

```
// cotuba.md.RenderizadorMDParaHTML

package cotuba.md;

import java.nio.file.Path;
import java.util.List;

import cotuba.domain.Capitulo;

public interface RenderizadorMDParaHTML {

  List<Capitulo> renderiza(Path diretorioDosMD);

}

// cotuba.md.RenderizadorMDParaHTMLImpl

package cotuba.md;

// imports omitidos...

public class RenderizadorMDParaHTMLImpl
    implements RenderizadorMDParaHTML { // modificado
```

```
  @Override //inserido
  public List<Capitulo> renderiza(Path diretorioDosMD) {
    // código omitido...
  }
}

// cotuba.pdf.GeradorPDF

package cotuba.pdf;

import cotuba.domain.Ebook;

public interface GeradorPDF {

  void gera(Ebook ebook);

}

// cotuba.pdf.GeradorPDFImpl

package cotuba.pdf;

// imports omitidos...

public class GeradorPDFImpl implements GeradorPDF { // modificado

  @Override //inserido
  public void gera(Ebook ebook) {
    // código omitido...
  }
}

// cotuba.epub.GeradorEPUB

package cotuba.epub;

import cotuba.domain.Ebook;

public interface GeradorEPUB {

  void gera(Ebook ebook);
```

```
}

// cotuba.epub.GeradorEPUBImpl

package cotuba.epub;

// imports omitidos...

public class GeradorEPUBImpl
    implements GeradorEPUB { // modificado

  @Override //inserido
  public void gera(Ebook ebook) {
    // código omitido...
  }
}
```

Na classe `Cotuba`, devemos depender o máximo possível das interfaces.

```
package cotuba.application;

// outros imports...
import cotuba.epub.GeradorEPUBImpl; // inserido
import cotuba.md.RenderizadorMDParaHTMLImpl; // inserido
import cotuba.pdf.GeradorPDFImpl; // inserido

public class Cotuba {

  public void executa(String formato, Path diretorioDosMD,
      Path arquivoDeSaida) {

    RenderizadorMDParaHTML renderizador =
        new RenderizadorMDParaHTMLImpl(); // modificado
    List<Capitulo> capitulos =
        renderizador.renderiza(diretorioDosMD);

    // código omitido...

    if ("pdf".equals(formato)) {
```

```
        GeradorPDF geradorPDF =
            new GeradorPDFImpl(); // modificado
        geradorPDF.gera(ebook);

      } else if ("epub".equals(formato)) {

        GeradorEPUB geradorEPUB =
            new GeradorEPUBImpl(); // modificado
        geradorEPUB.gera(ebook);

      } else {
        throw new IllegalArgumentException(
              "Formato do ebook inválido: " + formato);
      }
    }

}
```

4.7 Design Pattern: Factory

Repare na seguinte linha da classe `Cotuba`:

```
RenderizadorMDParaHTML renderizador = new RenderizadorMDParaHTMLImpl();
```

Depender da abstração oferecida pela interface `RenderizadorMDParaHTML` é exatamente o que queremos. Porém, ao instanciar `RenderizadorMDParaHTMLImpl`, também estamos dependendo da implementação dessa abstração. Isso é uma violação do DIP: além da abstração, um código de alto nível acabou dependendo diretamente de uma classe de baixo nível.

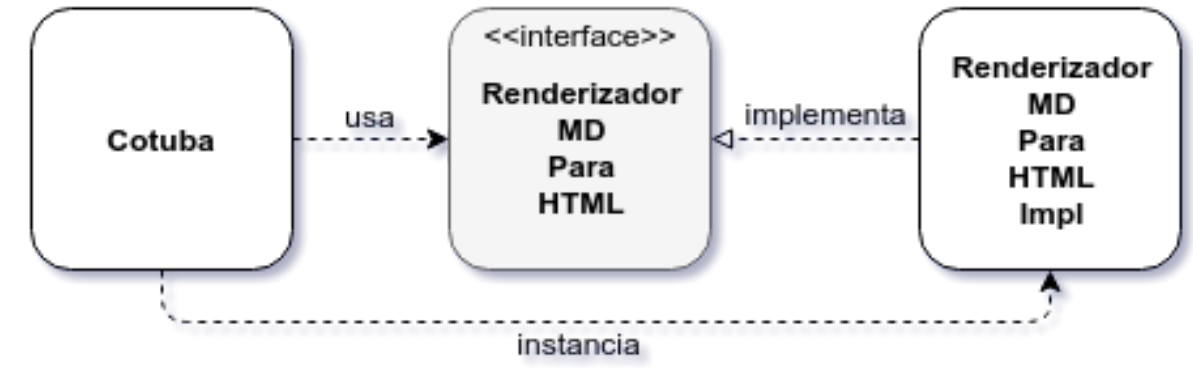

Figura 4.4: Cotuba instanciando implementação.

> *Um dos lugares mais comuns em que um design de código depende de classes concretas é ao criar instâncias. Por definição, não é possível instanciar abstrações. Portanto, para criar instâncias, é preciso depender de classes concretas.*
>
> Uncle Bob, no artigo *Design Principles and Design Patterns* (MARTIN, 2000)

Instanciar objetos é um problema comum em OO. Há um capítulo do livro *Design Patterns* (GAMMA et al., 1994) que cataloga diversos *Creational Patterns*, que são soluções para a criação de objetos como *Abstract Factory*, *Builder*, *Factory Method*, *Prototype* e *Singleton*.

Uma das responsabilidades recorrentes descritas por Craig Larman nos GRASP patterns do livro *Applying UML and Patterns* (LARMAN, 2004) é o *Creator*, um tipo de objeto que encapsula a criação de outros objetos. Larman diz que uma solução extremamente difundida para a criação de objetos é uma simplificação da *Abstract Factory*, que o autor chama de *Simple Factory*, *Concrete Factory* ou simplesmente **Factory**.

> *"Uma Factory tem várias vantagens:*
>
> - *Separa a responsabilidade da criação de objetos complexos em objetos auxiliares coesos.*
> - *Esconde lógica de instanciação potencialmente complexa.*
> - *Permite a introdução de estratégias de gerenciamento de memória que melhoram o desempenho, como cache ou reciclagem de objetos."*
>
> Craig Larman, no livro *Applying UML and Patterns* (LARMAN, 2004).

Explorando o código de uma Factory

Uma maneira comum de criar uma *Factory* em Java é ter um método estático que, quando chamado, retorna uma nova instância da implementação "escondida" por trás de uma abstração.

Em geral, colocamos um sufixo `Factory` na nova classe:

```
public class RenderizadorMDParaHTMLFactory {

  public static RenderizadorMDParaHTML cria() {
```

```
    return new RenderizadorMDParaHTMLImpl();
  }

}
```

Um fato interessante é que, a partir do Java 8, podemos ter *métodos estáticos em interfaces*. Portanto, podemos evitar a criação de mais uma classe colocando o método `cria` na própria interface `RenderizadorMDParaHTML`:

```
// cotuba.md.RenderizadorMDParaHTML

public interface RenderizadorMDParaHTML {

  List<Capitulo> renderiza(Path diretorioDosMD);

  // inserido
  static RenderizadorMDParaHTML cria() {
    return new RenderizadorMDParaHTMLImpl();
  }

}
```

> Observação: como todo método de uma interface é público, mesmo sendo estático, podemos omitir o modificador de acesso `public`.

Devemos também definir métodos estáticos que servirão como `Factory` para as implementações das interfaces `GeradorPDF` e `GeradorEPUB`, além de `RenderizadorMDParaHTML`. Em seguida, precisamos usar esses novos métodos na classe `Cotuba`.

Vamos definir o método `cria` na interface `GeradorPDF`, retornando um `GeradorPDFImpl`:

```
// cotuba.pdf.GeradorPDF

public interface GeradorPDF {
```

```
  void gera(Ebook ebook);

  // inserido
  static GeradorPDF cria() {
    return new GeradorPDFImpl();
  }

}
```

Devemos fazer o mesmo para a interface `GeradorEPUB`:

```
// cotuba.epub.GeradorEPUB

public interface GeradorEPUB {

  void gera(Ebook ebook);

  // inserido
  static GeradorEPUB cria() {
    return new GeradorEPUBImpl();
  }

}
```

Na classe `Cotuba`, vamos trocar a instanciação de `RenderizadorMDParaHTMLImpl` pela chamada do método `cria` da interface `RenderizadorMDParaHTML`:

```
// cotuba.application.Cotuba

RenderizadorMDParaHTML renderizador =
    new RenderizadorMDParaHTMLImpl();
RenderizadorMDParaHTML renderizador =
    RenderizadorMDParaHTML.cria();
```

O mesmo deve ser feito para a interface `GeradorPDF`:

```
// cotuba.application.Cotuba

GeradorPDF renderizador = new GeradorPDFImpl();
GeradorPDF geradorPDF = GeradorPDF.cria();
```

Finalmente, vamos usar o método `cria` da interface `GeradorEPUB`:

```
// cotuba.application.Cotuba

GeradorEPUB geradorEPUB = new GeradorEPUBImpl();
GeradorEPUB geradorEPUB = GeradorEPUB.cria();
```

Então, podemos remover os imports referentes às implementações `RenderizadorMDParaHTMLImpl`, `GeradorPDFImpl` e `GeradorEPUBImpl`. Ao testarmos a geração de PDF e EPUB, tudo deve funcionar!

Pronto! Agora `Cotuba`, uma classe de alto nível, depende apenas de abstrações! A criação dos objetos ficou escondida em cada Factory.

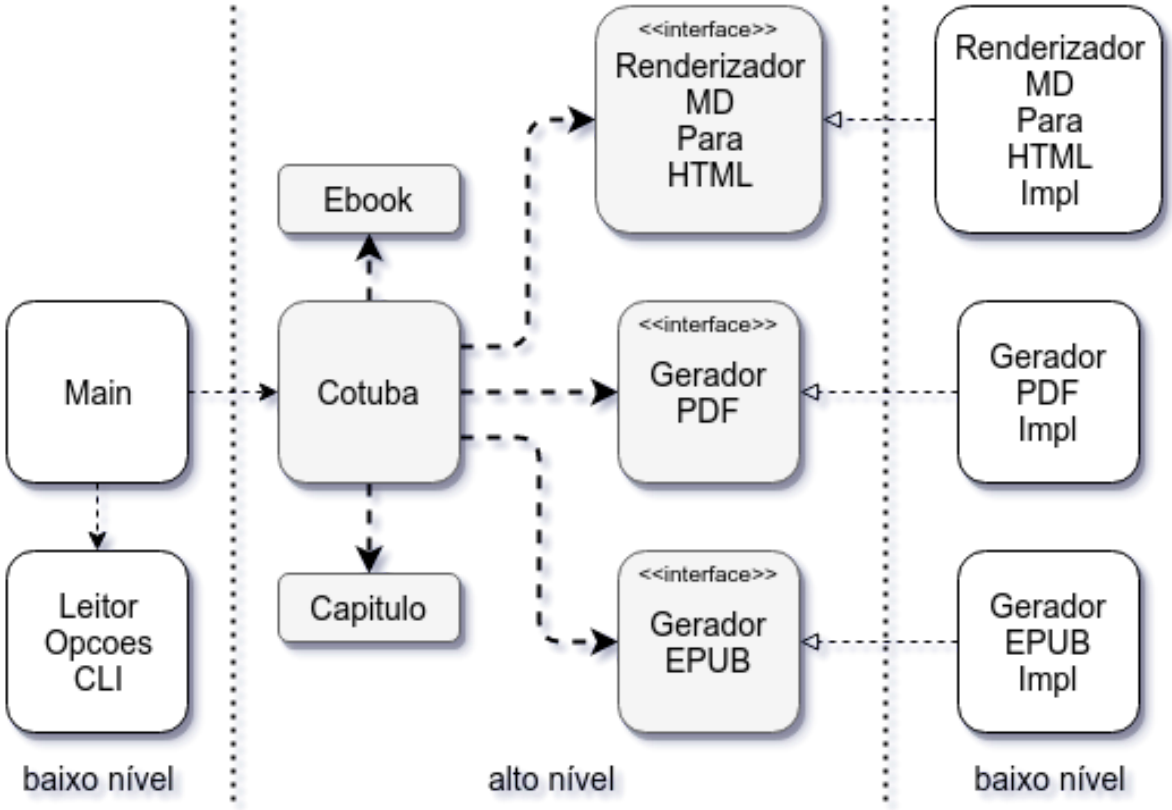

Figura 4.5: Dependências do Cotuba invertidas.

Discussão: qual a tradução de Design Patterns?

O título da tradução em português do livro *Design Patterns* (GAMMA et al., 1994) é *Padrões de Projeto*. Será que essa é uma boa tradução? Não, já que pode levar a uma confusão de conceitos. E essa confusão acontece porque as palavras "Padrão" e "Projeto" têm vários significados na língua portuguesa.

Em português, a palavra "Padrão" pode significar duas coisas:

- *Norma*: conjunto de regras de execução. Por exemplo, o padrão brasileiro de tomadas, definido pela ABNT. É o sentido mais comum em português.
- *Molde*: conjunto de características que identificam algo. Por exemplo, os padrões quadriculados de clãs escoceses (e das suas saias). É o sentido mais comum em inglês.

A palavra "Projeto" também tem dois significados em português:

- *Plano*: esquema detalhado de realização de um empreendimento, que envolve cronograma, prazo, custo, escopo, e mais. É algo que os gerentes de projeto tentam fazer.
- *Modelagem*: o ato de projetar, de delinear algo intelectualmente, geralmente acomodando-o a um molde. Algo que paisagistas, urbanistas e designer gráficos fazem.

Os Design Patterns do mundo do software são uma coleção de soluções comuns para problemas recorrentes. Não são normas de planos, nem moldes de planos. Também não são normas de modelagem. Design patterns são **moldes de modelagem**. Essa seria uma tradução mais adequada.

4.8 Melhorando o nome das implementações

Em projetos Java, é comum colocar o sufixo `Impl` na classe que implementa uma interface quando há uma única implementação. Por exemplo, para a interface `EnviadorSMS`, teríamos a classe `EnviadorSMSImpl` como implementação.

Não soa estranho ter classes como `EnviadorSMSImpl`? Uma alternativa é colocar no nome da implementação algo que revela a tecnologia usada.

No caso da interface `EnviadorSMS`, como usamos a API do Twilio, a implementação poderia ser chamada de `EnviadorSMSComTwilio`. Se modificarmos a implementação para usar a API do Nexmo, poderíamos chamá-la de `EnviadorSMSComNexmo`.

De maneira análoga, teríamos:

- `EnviadorEmailComMailgun` em vez de `EnviadorEmailImpl`.
- `NotaFiscalDAOComHibernate` em vez de `NotaFiscalDAOImpl`.

Melhorando o nome das implementações no Cotuba

Vamos remover o sufixo `Impl` das implementações, renomeando-as e considerando que:

- A renderização de Markdown usa a biblioteca CommonMark Java.
- A geração de PDF usa a biblioteca iText.
- A geração de EPUB usa a biblioteca Epublib.

Por isso, vamos renomear as implementações das dependências de baixo nível de `Cotuba` da seguinte maneira:

- De `RenderizadorMDParaHTMLImpl` para `RenderizadorMDParaHTMLComCommonMark`.
- De `GeradorPDFImpl` para `GeradorPDFComIText`.
- De `GeradorEPUBImpl` para `GeradorEPUBComEpublib`.

4.9 Abstrações mais perto de quem as utiliza

As interfaces `RenderizadorMDParaHTML`, `GeradorPDF` e `GeradorEPUB`, que servem como abstrações para as dependências de baixo nível da classe `Cotuba`, estão próximas às suas implementações:

- A interface `RenderizadorMDParaHTML` está no pacote `cotuba.md`.
- A interface `GeradorPDF` está no pacote `cotuba.pdf`.
- A interface `GeradorEPUB` está no pacote `cotuba.epub`.

Parece bem organizado mas há um problema de design. Para revelar esse problema, vamos pensar nas dependências um nível acima das classes: as **dependências entre pacotes**.

A classe `Cotuba` é a *cliente* das interfaces `RenderizadorMDParaHTML`, `GeradorPDF` e `GeradorEPUB`. Afinal de contas, são abstrações criadas exatamente para inverter as dependências de `Cotuba`. Veja os imports:

```
// cotuba.application.Cotuba

package cotuba.application;

// outros imports...
import cotuba.epub.GeradorEPUB;
import cotuba.md.RenderizadorMDParaHTML;
import cotuba.pdf.GeradorPDF;

public class Cotuba {

  // código omitido...

}
```

O pacote de `Cotuba` é `cotuba.application`, que acaba dependendo de abstrações dos pacotes `cotuba.md`, `cotuba.pdf` e `cotuba.epub`.

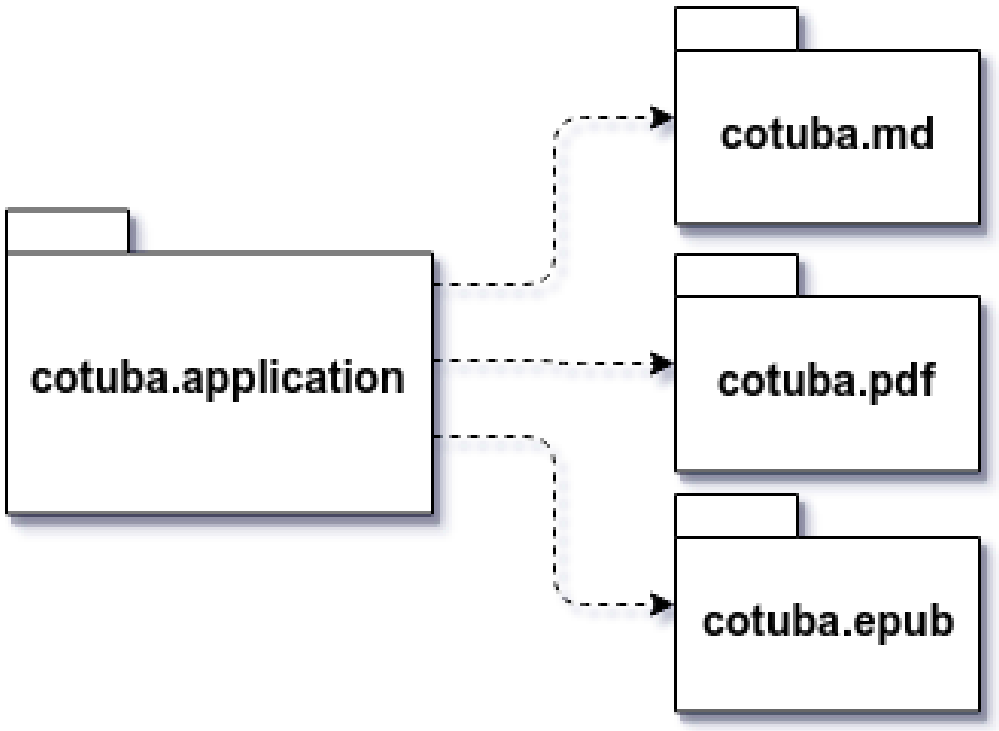

Figura 4.6: Dependências entre pacotes.

Um pacote com código de alto nível está dependente de pacotes com código de baixo nível. Isso fere o DIP!

Agora, se movermos as interfaces usadas por `Cotuba` para o pacote `cotuba.application`, as abstrações estarão mais próximas de quem as utiliza. A dependência entre os pacotes também passa a ser invertida: o pacote de alto nível não depende mais dos pacotes de baixo nível, mas o contrário!

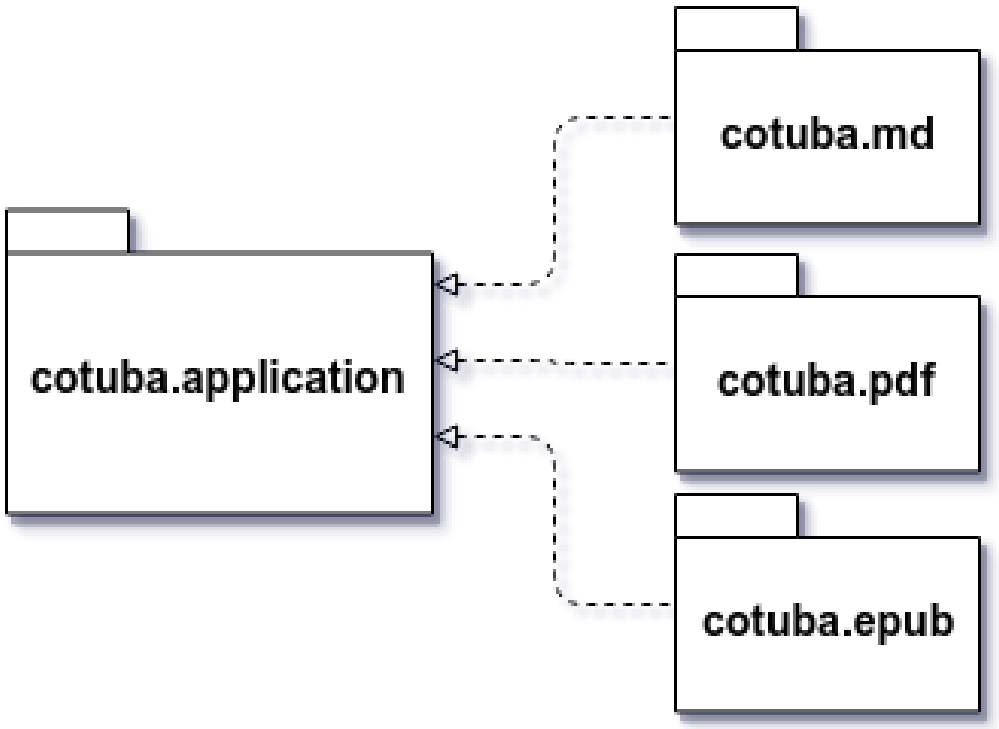

Figura 4.7: Dependências entre pacotes invertidas.

> *"Interfaces de serviço, em geral, são do cliente (quem as usa)."*
>
> Uncle Bob, no livro *Agile Principles, Patterns, and Practices in C#* (MARTIN, 2006).

Movendo as abstrações do Cotuba

Vamos mover as interfaces `RenderizadorMDParaHTML`, `GeradorPDF` e `GeradorEPUB` para o pacote `cotuba.application`.

```
cotuba
└── application
    ├── Cotuba.java
    ├── RenderizadorMDParaHTML.java
    ├── GeradorPDF.java
    └── GeradorEPUB.java
```

Como o pacote dessas interfaces será o mesmo da classe `Cotuba`, os imports dessas interfaces podem ser removidos.

Discussão

Será mesmo que invertemos as dependências dos pacotes? Qual detalhe ainda faz com que, pensando em pacotes, haja uma quebra do DIP?

A questão aqui é que as *factories* dependem de suas respectivas implementações. Para resolver, poderíamos:

- Usar algum framework de Dependency Injection, como o Spring.
- Receber a classe da implementação por uma propriedade ou variável de ambiente e usar Reflection para instanciá-la, o que é sugerido por Craig Larman no livro *Applying UML and Patterns* (LARMAN, 2004).
- Usar o Service Loader API do próprio Java (o que faremos em um capítulo posterior).

4.10 Uma classe para os parâmetros

Repare no método `executa` da classe `Cotuba`:

```
public class Cotuba {

  public void executa(String formato, Path diretorioDosMD,
      Path arquivoDeSaida) {
    // código omitido...
  }

}
```

Esse método recebe três parâmetros atualmente. A chance de esse número aumentar é alta. Em Java, uma lista longa de parâmetros é algo terrível de entender, principalmente na chamada do método.

Poderíamos criar uma classe para agrupar esses parâmetros, simplificando a assinatura do método `executa`. O livro *Refactoring* (FOWLER et al., 1999) cataloga a refatoração *Introduce Parameter Object* (Introduzir Objeto Parâmetro).

Além de simplificar a assinatura de um método, outra motivação dessa refatoração é agrupar parâmetros que aparecem juntos, repetidamente, em diversos métodos da mesma classe ou de classes diferentes.

Uma das vantagens é ter um lugar para colocar a lógica que manipula esses parâmetros. Mas veja só, já temos uma classe que agrupa `formato`, `diretorioDosMD` e `arquivoDeSaida`: a classe `LeitorOpcoesCLI`!

Simplificando parâmetros

Vamos alterar o método `executa` da classe `Cotuba` para receber um `LeitorOpcoesCLI` como parâmetro:

```
// cotuba.application.Cotuba

public class Cotuba {

  public void executa(LeitorOpcoesCLI parametros) { // modificado

    // restante do código ...
```

Não podemos nos esquecer do import da classe `LeitorOpcoesCLI` já que está em outro pacote, o `cotuba.cli` . Teríamos que alterar o modificador de acesso dessa classe para `public` , para que seja acessível por classes de outros pacotes:

```
// cotuba.cli.LeitorOpcoesCLI

public class LeitorOpcoesCLI { // modificado
```

Neste momento, o código deve apresentar erros de compilação no uso dos parâmetros antigos. Vamos corrigi-los no próximo passo. Dentro do método `executa` de `Cotuba` , devemos criar novas variáveis com os mesmos nomes dos parâmetros que foram removidos, `formato` , `diretorioDosMD` e `arquivoDeSaida` :

```
// cotuba.application.Cotuba

public void executa(LeitorOpcoesCLI parametros) {

  // inserido
  String formato = parametros.getFormato();
  Path diretorioDosMD = parametros.getDiretorioDosMD();
  Path arquivoDeSaida = parametros.getArquivoDeSaida();

  // restante do código ...
```

Agora o código da classe `Cotuba` deve ser compilado com sucesso!

Na classe `Main`, precisamos alterar a chamada do método `executa` da classe `Cotuba`, passando o objeto `opcoesCLI`. Vamos remover as variáveis desnecessárias, mantendo as que ainda são usadas.

```
// cotuba.cli.Main

public class Main {

  public static void main(String[] args) {

    Path diretorioDosMD;
    String formato;
    Path arquivoDeSaida;
    boolean modoVerboso = false;

    try {

      var opcoesCLI = new LeitorOpcoesCLI(args);

      diretorioDosMD = opcoesCLI.getDiretorioDosMD();
      formato = opcoesCLI.getFormato();
      arquivoDeSaida = opcoesCLI.getArquivoDeSaida();
      modoVerboso = opcoesCLI.isModoVerboso();

      try {

        var cotuba = new Cotuba();

        cotuba.executa(formato, diretorioDosMD, arquivoDeSaida);
        cotuba.executa(opcoesCLI);
```

Ao testarmos a geração do PDF e do EPUB, tudo deve funcionar!

Ih! Acabamos de ferir o DIP...

A classe `Cotuba`, do pacote `cotuba.application`, é uma classe relacionada à aplicação, de alto nível, que tem a ver com o problema que estamos resolvendo. Já a classe `LeitorOpcoesCLI`, do pacote `cotuba.cli`, é um detalhe de implementação, de baixo nível,

que tem a ver com o mecanismo de entrega de UI que escolhemos: a linha de comando.

Ao fazermos `Cotuba` receber um objeto da classe `LeitorOpcoesCLI` como parâmetro em seu método `executa`, fizemos um código de alto nível depender de um de baixo nível. Por isso, violamos o DIP.

Como fazer para aderir ao DIP novamente? Criar uma abstração, invertendo a dependência!

Mais uma interface para inverter as dependências

Vamos criar uma abstração chamada `ParametrosCotuba` a partir de `LeitorOpcoesCLI` e usá-la como parâmetro do método `executa` da classe `Cotuba`. Para isso, vamos extrair uma interface `ParametrosCotuba` da classe que contém *getters* para os atributos `formato`, `diretorioDosMD` e `arquivoDeSaida`.

Devemos nos certificar de que a nova interface esteja declarada no pacote `cotuba.application`, a fim de respeitar o DIP no nível de pacotes.

> Dica: usa a ajuda da sua IDE para extrair a nova interface!

A nossa nova abstração, a interface `ParametrosCotuba`, ficará semelhante a:

```
// cotuba.application.ParametrosCotuba

package cotuba.application;

import java.nio.file.Path;

public interface ParametrosCotuba {

  Path getDiretorioDosMD();

  String getFormato();
```

```
  Path getArquivoDeSaida();

}
```

A classe de baixo nível `LeitorOpcoesCLI` deverá implementar a nova abstração. Além disso, esta classe não precisa mais ser pública:

```
// cotuba.cli.LeitorOpcoesCLI

// outros imports...
import cotuba.application.ParametrosCotuba; // inserido

// modificado
public class LeitorOpcoesCLI implements ParametrosCotuba {

  // restante do código...
}
```

Na classe `Cotuba`, devemos usar a interface `ParametrosCotuba`, a fim de respeitar o DIP!

```
// cotuba.application.Cotuba

// outros imports...
import cotuba.cli.LeitorOpcoesCLI;

public class Cotuba {

  // modificado
  public void executa(ParametrosCotuba parametros) {

  // restante do código...
```

A classe `Cotuba` passa a depender apenas:

- De bibliotecas padrão do Java: API de Collections e NIO.
- Das classes de domínio: `Ebook` e `Capitulo`.
- De abstrações: `ParametrosCotuba`, `RenderizadorMDParaHTML`, `GeradorPDF` e `GeradorEPUB`.

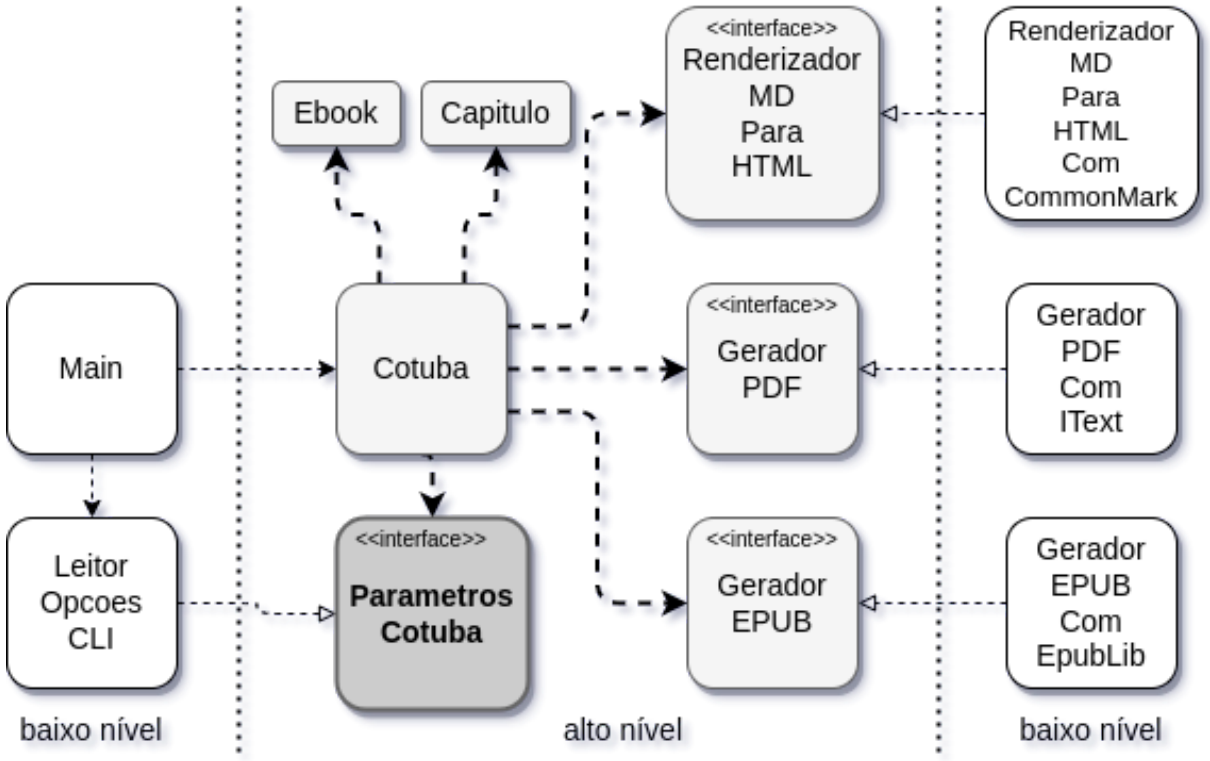

Figura 4.8: Dependências do Cotuba.

A geração dos PDFs e EPUBs deve continuar funcionando!

Para saber mais: Dependency Injection e Spring

DIP tem a ver com a qualidade das dependências: garantir que regras de negócio não dependam de detalhes de implementação. Já *Dependency Injection (DI)* está relacionado com a maneira como um objeto obtém as suas dependências.

Quando usamos DI, um framework ou contêiner (Guice, Spring ou outro) fornece instâncias das dependências para um determinado objeto, em vez de o próprio objeto buscar essas instâncias por meio de *Factories*. Por isso, podemos dizer que DI é uma alternativa à utilização de uma Factory.

Dependency Injection com o Spring

O Spring é um ecossistema de ferramentas para aumentar a produtividade de projetos Java. Mas sua história começou como um framework de DI. Para utilizá-lo especificamente para DI, devemos declarar um de seus subprojetos, o Spring Context, como dependência no Maven:

```
<!-- pom.xml -->

<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-context</artifactId>
  <version>5.3.15</version>
</dependency>
```

Então, podemos remover os métodos estáticos `cria` das interfaces `GeradorPDF`, `GeradorEPUB` e `RenderizadorMDParaHTML`, já que não precisaremos mais definir Factories.

Devemos configurar o Spring para que gerencie as implementações das interfaces mencionadas. Nas primeiras versões do Spring, teríamos que fazer essas configurações por meio de arquivos XML. A partir do Spring 2.5, podemos usar uma anotação como `@Component` ou outras com um significado mais específico como `@Controller`, `@Service` e `@Repository`.

Vamos anotar a classe `GeradorPDFComIText` com `@Component`:

```
// cotuba.pdf.GeradorPDFComIText

@Component // inserido
public class GeradorPDFComIText
    implements GeradorPDF {
```

Deve ser adicionado o import à anotação `org.springframework.stereotype.Component`. O mesmo deve ser feito para as outras classes de implementação: `GeradorEPUBComEpublib` e `RenderizadorDeMDParaHTMLComCommonMark`. Também devemos anotar a classe `Cotuba` com `@Component`.

Além disso, devemos remover de `Cotuba` as chamadas às Factories. No lugar, devemos definir atributos para `GeradorPDF`, `GeradorEPUB` e `RenderizadorMDParaHTML`. Além disso, deve ser definido um construtor que recebe instâncias dessas interfaces. As implementações gerenciadas pelo Spring serão injetadas pelo construtor.

> Em versões anteriores do Spring, seria necessário anotar o construtor com `@Autowired`. Mas desde a versão 4.3, essa anotação não é mais necessária.

```
// cotuba.application.Cotuba

@Component // anotação adicionada
public class Cotuba {

  // atributos inseridos
  private final GeradorPDF geradorPDF;
  private final GeradorEPUB geradorEPUB;
  private final RenderizadorMDParaHTML renderizador;

  // recebendo instâncias pelo construtor
  public Cotuba(GeradorPDF geradorPDF,
    GeradorEPUB geradorEPUB,
    RenderizadorMDParaHTML renderizador) {
      this.geradorPDF = geradorPDF;
      this.geradorEPUB = geradorEPUB;
      this.renderizador = renderizador;
  }

  public void executa(ParametrosCotuba parametros) {

    // código omitido ...

    // factory removida
    RenderizadorMDParaHTML renderizador = RenderizadorMDParaHTML.cria();
    List<Capitulo> capitulos =
        renderizador.renderiza(diretorioDosMD);
```

```
    // código omitido ...

    if ("pdf".equals(formato)) {

      // factory removida
      GeradorPDF geradorPDF = GeradorPDF.cria();

      geradorPDF.gera(ebook);

    } else if ("epub".equals(formato)) {

      // factory removida
      GeradorEPUB geradorEPUB = GeradorEPUB.cria();

      geradorEPUB.gera(ebook);
    }
    // restante do código ...
```

Além disso, devemos remover os métodos `cria` das interfaces `GeradorPDF`, `GeradorEPUB` e `RenderizadorMDParaHTML`.

Devemos criar uma configuração para que o Spring encontre os componentes (ou *beans*) gerenciados. Para isso, vamos definir uma classe `CotubaConfig` em um pacote acima dos demais. O Spring vasculhará todos os pacotes abaixo.

```
// cotuba.CotubaConfig

package cotuba;

import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;

@Configuration
@ComponentScan
public class CotubaConfig {
}
```

Na classe `Main`, devemos usar um `ApplicationContext` do Spring para obter uma instância de `Cotuba` com todas as dependências já

injetadas:

```
// cotuba.cli.Main

Cotuba cotuba = new Cotuba();

// inserido
ApplicationContext applicationContext =
    new AnnotationConfigApplicationContext(CotubaConfig.class);
Cotuba cotuba = applicationContext.getBean(Cotuba.class);

cotuba.executa(opcoesCLI);
```

Devem ser feitos novos imports:

```
// cotuba.cli.Main

import org.springframework.context.ApplicationContext;
import
org.springframework.context.annotation.AnnotationConfigApplicationContext;
```

Pronto! A geração de PDFs e EPUBs deve continuar funcionando!

O código anterior pode ser encontrado em: https://github.com/alexandreaquiles/solid-na-pratica/tree/cap4-extra-dependency-injection-com-spring

Para saber mais: invertendo dependências com o Observer pattern

Imagine um alerta que é executado quando um botão é clicado. Seria muito ruim, em termos de performance, se o código do alerta tivesse que perguntar de tempos em tempos se o botão já foi clicado. Seria algo semelhante ao burrinho no filme Shrek que, durante uma longa viagem, pergunta incessantemente: "Já chegamos lá?".

É muito comum em código de UI que trabalhemos com eventos. Em vez de perguntar se o botão já foi clicado, o código se registra em um evento de clique de um botão. É assim no JavaScript no navegador e também na programação Desktop em Java. Um exemplo com Swing:

```
var botao = new JButton("Clique aqui");
botao.addActionListener(evento -> System.out.println("Botão clicado!"));
```

Essa solução pode ser generalizada em um Design Pattern: o *Observer*. Os objetos são separados em:

- O *publisher*, que publica um evento.
- Os *subscribers*, interessados em executar ações quando o evento ocorre.

O *Observer pattern* também pode ser considerado um mecanismo para inverter dependências.

Pense no exemplo do começo do capítulo. Em vez de o `EmissorNotaFiscal` pedir para que o `EnviadorSMS` envie a nota por SMS, o emissor de notas fiscais poderia publicar um evento de `NotaFiscalEmitida`, que seria recebido pelo `EnviadorSMS`. Para isso, precisaríamos de alguma ferramenta que cuidasse dos eventos, dos publishers e dos subscribers. Seria uma outra maneira de inverter a dependência, evitando que código de alto nível dependa de código de baixo nível e, portanto, impedindo uma violação do DIP.

O uso de eventos é bastante feito em linguagens dinâmicas como Ruby para aderir ao DIP. No artigo *Under Deconstruction: The State of Shopify's Monolith* (MÜLLER, 2020), Philip Müller, engenheiro da Shopify, menciona que usa mecanismos do Rails para inverter dependências: *"(...) inverter uma dependência de maneira que o controle de fluxo e a dependência do código-fonte são opostas. Isso pode ser feito por meio de um mecanismo de publish/subscribe como `ActiveSupport::Notifications`."*

É possível utilizar eventos, publishers e subscribers para aplicar o **DIP na integração de sistemas distribuídos**. Um serviço de emissão de notas fiscais que dependesse diretamente de um serviço de envio de SMS poderia ter a dependência invertida se fosse usado um sistema de mensageria como RabbitMQ, Apache Kafka ou NATS.

Para saber mais: Hollywood Principle e Inversion of Control (IoC)

Há milhares de atores e atrizes tentando a sorte em Hollywood. E todo filme tem uma pessoa responsável por definir quem vai interpretar cada papel. Imagine se os milhares de atores e atrizes ligassem à procura de uma oportunidade. A pessoa responsável pelo elenco não sairia do telefone!

Por isso, em Hollywood, a pessoa responsável pelo elenco obtém uma lista de atores e atrizes com seus telefones e as características de cada um. E essa pessoa comumente diz: "Não me ligue, deixa que eu te ligo". Esse é o *Hollywood Principle*.

No Design de Código, podemos organizar nosso código de acordo com o Hollywood Principle. Perceba que, ao usarmos Dependency Injection (DI), um objeto recebe suas dependências "de fora" em vez de buscá-las. Ao usar o Observer pattern, um subscriber recebe um evento, em vez de ficar perguntando se já aconteceu.

Há algo comum entre as duas soluções, que é o que chamamos de *Inversion of Control (IoC)*, ou inversão de controle: é o Hollywood Principle aplicado a código.

4.11 Contraponto: críticas ao DIP

No livro *SOLID is not solid* (COPELAND, 2019), David Copeland critica o artigo *Dependency Inversion Principle* (MARTIN, 1996c), em que há a argumentação inicial em torno do DIP. Nesse artigo, Uncle Bob argumenta pelo DIP usando como exemplo um programa que envia o que é digitado em um teclado para uma impressora e que precisa ser adaptado para enviar os dados para um arquivo. Segundo Copeland, é um exemplo fabricado, longe do dia a dia de quem desenvolve. Além disso, o exemplo usa I/O, que é um problema já resolvido na maioria das linguagens de programação. Há ainda uma crítica à promessa de flexibilidade e reúso que, de acordo com Copeland, leva a códigos desnecessariamente complexos.

No artigo *CUPID - the back story* (NORTH, 2021), Dan North argumenta que a obsessão com inversão de dependências causou bilhões de dólares em custos desnecessários. Segundo North, o princípio só deveria ser usado quando há múltiplas maneiras de prover uma dependência e que a maioria das dependências não precisam de opções. North diz ainda que há bases de código em que toda classe implementa exatamente uma interface apenas para satisfazer um framework de injeção de dependências ou de testes automatizados. Também há a argumentação de que a promessa de poder trocar uma dependência complexa como um banco de dados evapora assim que tentamos realizar essa tarefa na prática.

4.12 O que aprendemos?

Neste capítulo, estudamos as vantagens de depender de abstrações e como devemos usá-las para fazer com que código alinhado às regras de negócio (o código de alto nível) não dependa diretamente dos detalhes de implementação (o código de baixo nível). Ao organizarmos o código dessa maneira, estaremos seguindo o DIP e, por consequência, teremos dependências mais estáveis!

No próximo capítulo, veremos como seguir o Princípio Aberto/Fechado para criarmos boas abstrações que favoreçam a flexibilidade do nosso código!

O código deste capítulo pode ser encontrado em: https://github.com/alexandreaquiles/solid-na-pratica/tree/cap4-dependency-inversion-principle

Figura 4.9: Vídeo do capítulo

CAPÍTULO 5

Princípio Aberto/Fechado (OCP): objetos flexíveis

No capítulo anterior, classificamos o Cotuba em código de alto nível, alinhado às regras de negócio, e de baixo nível, relacionado a detalhes de implementação como geração de PDFs, EPUBs e renderização de Markdown. Nos pontos em que código de alto nível dependia diretamente de código de baixo nível, usamos abstrações e fizemos com que as dependências fossem invertidas, seguindo o DIP.

Note que cada uma das abstrações criadas no capítulo anterior tem exatamente uma implementação:

- A interface `GeradorPDF` é implementada por `GeradorPDFComIText`.
- A interface `GeradorEPUB` é implementada por `GeradorEPUBComEpublib`.
- A interface `RenderizadorMDParaHTML` é implementada por `RenderizadorMDParaHTMLComCommonMark`.
- A interface `ParametrosCotuba` é implementada por `LeitorOpcoesCLI`.

Não é estranho que toda abstração tenha somente uma implementação? Talvez isso seja uma indicação de que podemos repensar a organização do nosso código e criar abstrações melhores. É o que faremos neste capítulo!

5.1 Em busca da abstração perfeita

Vamos voltar ao exemplo do `EmissorNotaFiscal` que vimos no capítulo sobre DIP:

```
public class EmissorNotaFiscal {

  private RegrasDeTributacao tributacao;
  private LegislacaoFiscal legislacao;
  private NotaFiscalDAO notas;
  private EnviadorEmail email;
  private EnviadorSMS sms;

  //...

  public NotaFiscal gera(Fatura fatura) {

    List<Imposto> impostos = tributacao.verifica(fatura);
    List<Isencao> isencoes = legislacao.analisa(fatura);

    // método auxiliar
    NotaFiscal nota = aplica(impostos, isencoes);

    notas.salva(nota);

    String mensagemNota = String.format(
          "Sua nota fiscal foi gerada: %s",  nota.getNumero());

    email.envia(mensagemNota, fatura.getEmail());
    sms.envia(mensagemNota, fatura.getTelefone());

    return nota;
  }

}
```

Esse exemplo é inspirado no livro *OO e SOLID para Ninjas* (ANICHE, 2015).

Considerando que `NotaFiscalDAO`, `EnviadorEmail` e `EnviadorSMS` são interfaces que abstraem as dependências de baixo nível da classe `EmissorNotaFiscal`, será que chegamos a um bom design? Repare que, ao final do método `gera` de `EmissorNotaFiscal`, são feitas várias ações:

- Persistir a nota gerada no BD através de `NotaFiscalDAO` e sua implementação `NotaFiscalDAOComHibernate`.
- Enviar uma notificação por e-mail através de `EnviadorEmail` e sua implementação `EnviadorEmailComMailgun`.
- Enviar uma notificação por SMS através de `EnviadorSMS` e sua implementação `EnviadorSMSComTwilio`.

Podemos buscar abstrações melhores. Abstrações mais abstratas! E-mail e SMS são tipos de notificação, não é mesmo? Poderíamos criar uma abstração para `Notificacao`, que seria implementada tanto por `EnviadorEmailComMailgun` como por `EnviadorSMSComTwilio`:

```
public interface Notificacao {
  void notifica(NotaFiscal nota);
}
```

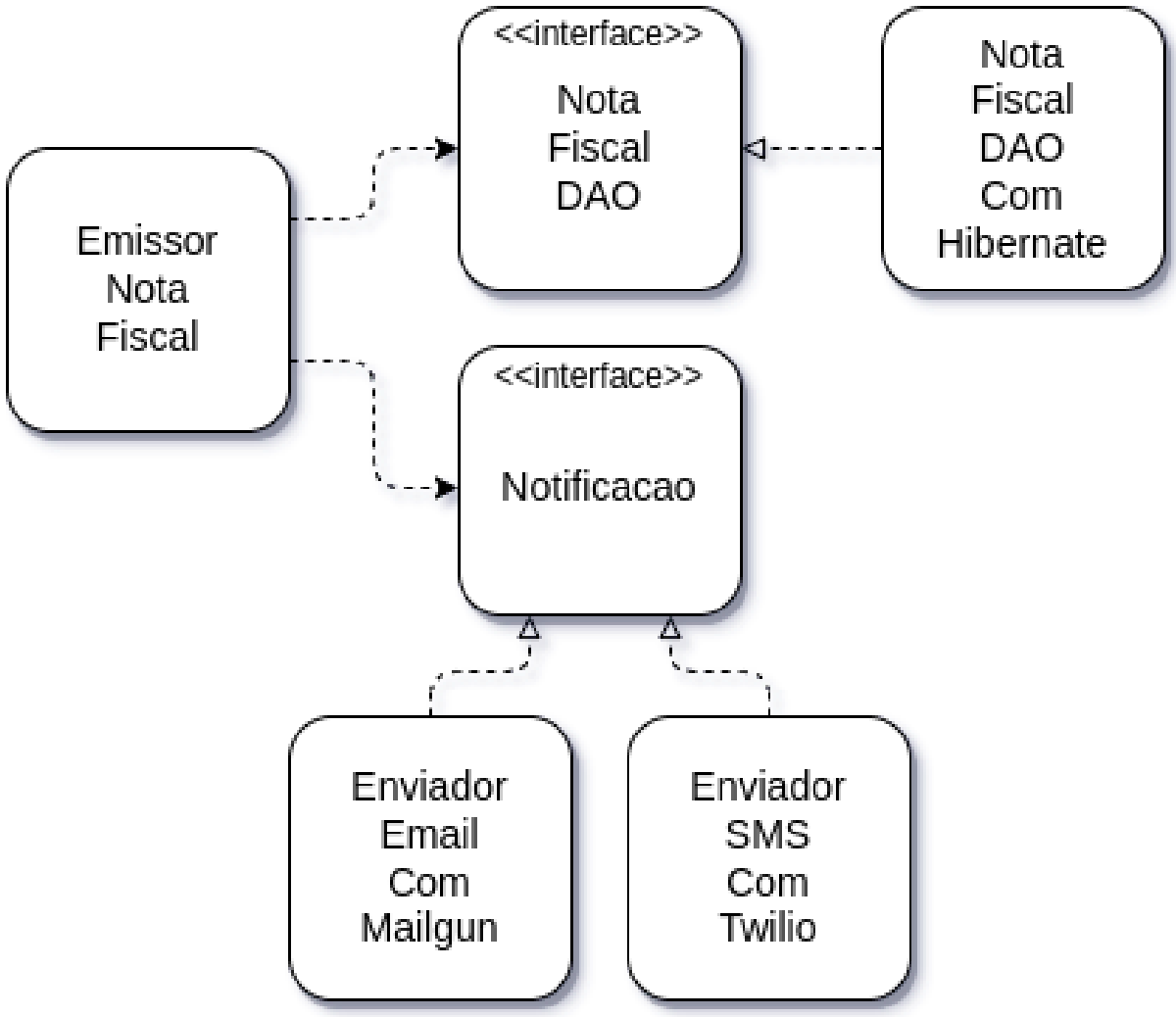

Figura 5.1: Notificação como abstração.

Mas podemos ir além: todas essas ações, de persistência ou de notificação, são um conjunto de ações que devem ser feitas após a emissão da nota fiscal. Que tal uma abstração chamada `AcaoPosEmissao`? Tanto `NotaFiscalDAOComHibernate` quanto `EnviadorEmailComMailgun` e `EnviadorSMSComTwilio` seriam implementações.

```
public interface AcaoPosEmissao {
  void faz(NotaFiscal nota);
}
```

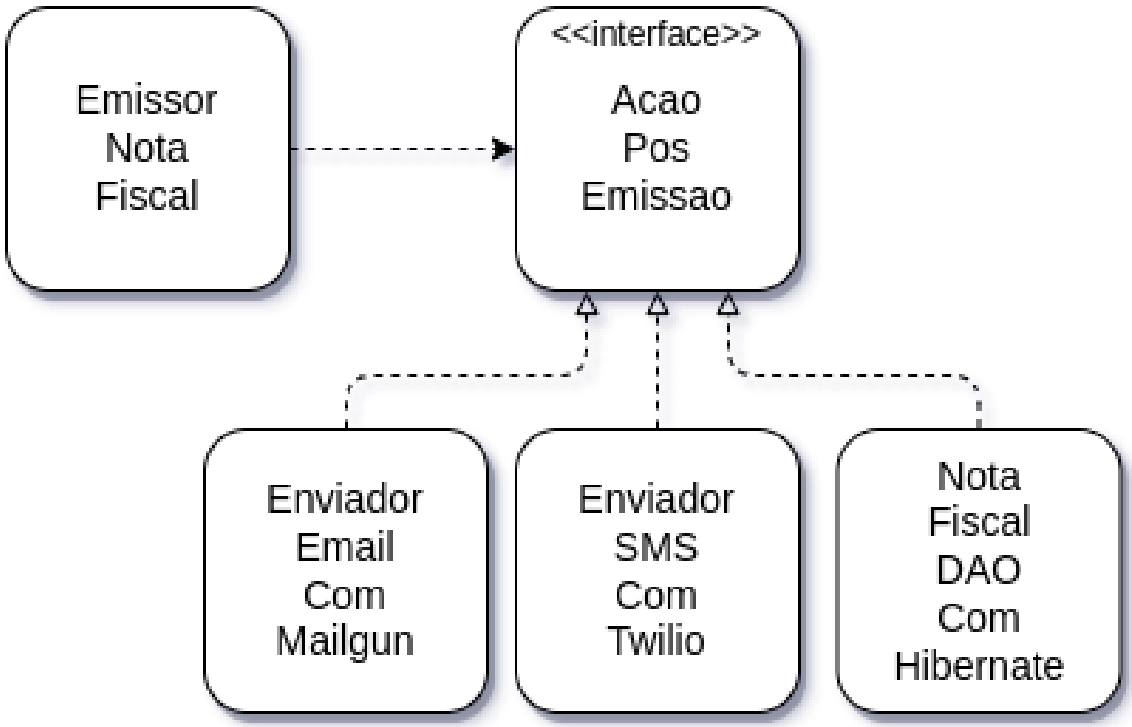

Figura 5.2: AcaoPosEmissao como abstração.

> Um exemplo de uma boa abstração é o mecanismo de I/O do Unix, que é composto por apenas 5 chamadas de sistema (ou *system calls*): `open`, `read`, `write`, `lseek` e `close`. Essa pequena interface permite que sejam lidos dados de fontes como teclados, arquivos, a rede e até da saída de outro programa. Também há flexibilidade na saída, que pode ter destinos como uma tela, arquivos, a rede ou outros programas.

Note que podemos encaixar qualquer tipo de ação na abstração `AcaoPosEmissao`. Ao surgirem novas necessidades, como envio para o SAP ERP ou para um novo sistema de uma prefeitura, bastaria fornecermos mais uma implementação.

Com uma abstração mais ampla, o código se torna extremamente **flexível**, resistindo a novas necessidades de negócio. É exatamente o que almejamos em um design OO.

FLEXÍVEL

Que se adapta bem a diferentes atividades ou funções; acomodatício, adaptável, moldável.

Fonte: Michaelis Online

Perceba que não há tanta diferença entre as interfaces `Notificacao` e `AcaoPosEmissao` em termos de código. Basicamente, o que muda são os nomes dos métodos e das próprias interfaces. A principal diferença é em termos *semânticos*: o significado da abstração muda, tornando-se mais moldável.

Qual devemos escolher? Depende da funcionalidade que queremos implementar. Código mais genérico é mais flexível mas, por outro lado, pode ser mais difícil de entender.

5.2 Design Pattern: Command

No código a seguir, adotamos a abstração de ações pós-emissão da nota fiscal. Na classe `EmissorNotaFiscal`, teríamos uma lista de `AcaoPosEmissao` que percorreríamos no final da geração da nota, invocando cada implementação:

```
public class EmissorNotaFiscal {

  private RegrasDeTributacao tributacao;
  private LegislacaoFiscal legislacao;
  private List<AcaoPosEmissao> acoes;

  //...

  public NotaFiscal gera(Fatura fatura) {

    List<Imposto> impostos = tributacao.verifica(fatura);
    List<Isencao> isencoes = legislacao.analisa(fatura);

    // método auxiliar
    NotaFiscal nota = aplica(impostos, isencoes);

    // modificado
    for (AcaoPosEmissao acao: acoes) {
      acao.faz(nota);
    }

    return nota;
  }

}
```

A abstração `AcaoPosEmissao` serve tanto para persistência, para notificações, como para qualquer outro tipo de operação a ser feita depois da emissão da nota fiscal.

A ideia de ter uma abstração que representa uma operação do sistema, sem detalhes sobre o que é feito, é descrita no **Command Pattern** do livro *Design Patterns* (GAMMA et al., 1994): *"Às vezes, é necessário fazer solicitações para objetos sem saber nada sobre a operação solicitada ou o destinatário da solicitação."*

Variações protegidas

Esconder os detalhes de várias implementações diferentes, mas relacionadas, por trás de uma mesma abstração é o que Craig Larman chama de *Variações Protegidas* no artigo *Protected Variation* (LARMAN, 2001): *"Identifique pontos previstos de variação e crie uma interface estável em torno deles."*

Precisamos, constantemente, analisar o nosso código em busca de pontos de articulação onde nosso design varia e, por meio de abstrações, torná-los flexíveis.

> *"Encontrar os pontos onde sua modelagem precisa ser flexível e onde não precisa é um desafio."*
>
> Maurício Aniche, no livro *OO e SOLID para Ninjas* (ANICHE, 2015)

5.3 O Princípio Aberto/Fechado (OCP)

Uncle Bob incluiu no SOLID um princípio relacionado à flexibilidade de objetos, originalmente cunhado pelo pioneiro da Orientação a Objetos Bertrand Meyer:

> **OPEN/CLOSED PRINCIPLE (OCP)**
>
> *Entidades de software (classes, módulos, funções etc.) devem ser abertas para extensão, mas fechadas para modificação.*

De acordo com o OCP, classes devem ser abertas e fechadas ao mesmo tempo. Como assim? Esse princípio trata de esclarecer que, em um bom design OO, deve ser possível criar novos comportamentos sem modificar o código já existente.

É o que fizemos ao criar abstração `AcaoPosEmissao` no exemplo anterior: podemos adicionar novas ações sem mudar a classe `EmissorNotaFiscal`, que dispara a execução dessas ações.

> *"Devemos escrever módulos que podem ser estendidos sem que sejam modificados. Mudar o que os módulos fazem sem mudar o seu código-fonte."*
>
> Uncle Bob, no artigo *Design Principles and Design Patterns* (MARTIN, 2000).

OCP e DIP

No mesmo artigo *Design Principles and Design Patterns*, Uncle Bob compara os princípios DIP e OCP, mostrando que são

complementares: *"Se o OCP declara o objetivo de uma arquitetura OO, o DIP declara o seu mecanismo fundamental."*

O DIP fomenta a classificação de dependências em alto/baixo nível e a inversão das dependências em direção às regras de negócio. Isso conduz a abstrações que levam a um código flexível, que pode ser estendido sem modificá-lo: o intuito do OCP.

5.4 Design Pattern: Strategy

Repare nas abstrações definidas pelas interfaces `GeradorPDF` e `GeradorEPUB`:

```
// cotuba.application.GeradorPDF

public interface GeradorPDF {

  void gera(Ebook ebook);

  // factory omitida...

}

// cotuba.application.GeradorEPUB

public interface GeradorEPUB {

  void gera(Ebook ebook);

  // factory omitida...

}
```

Ambas as interfaces definem um método `gera`, que recebe como parâmetro um `Ebook` e não tem nenhum retorno. Note que as abstrações são **idênticas**! Você deve estar se perguntando se há algo de errado nisso, não? Na verdade, não há propriamente um erro, mas há um ponto de melhoria.

Poderíamos criar uma abstração mais relacionada ao problema que estamos resolvendo: um gerador de ebooks. Esse gerador não definiria em qual formato o ebook deverá ser gerado. Esse detalhe é tarefa para as implementações. Assim, a geração de ebooks seria

mais flexível: para novos formatos, basta uma nova implementação. É o espírito do OCP!

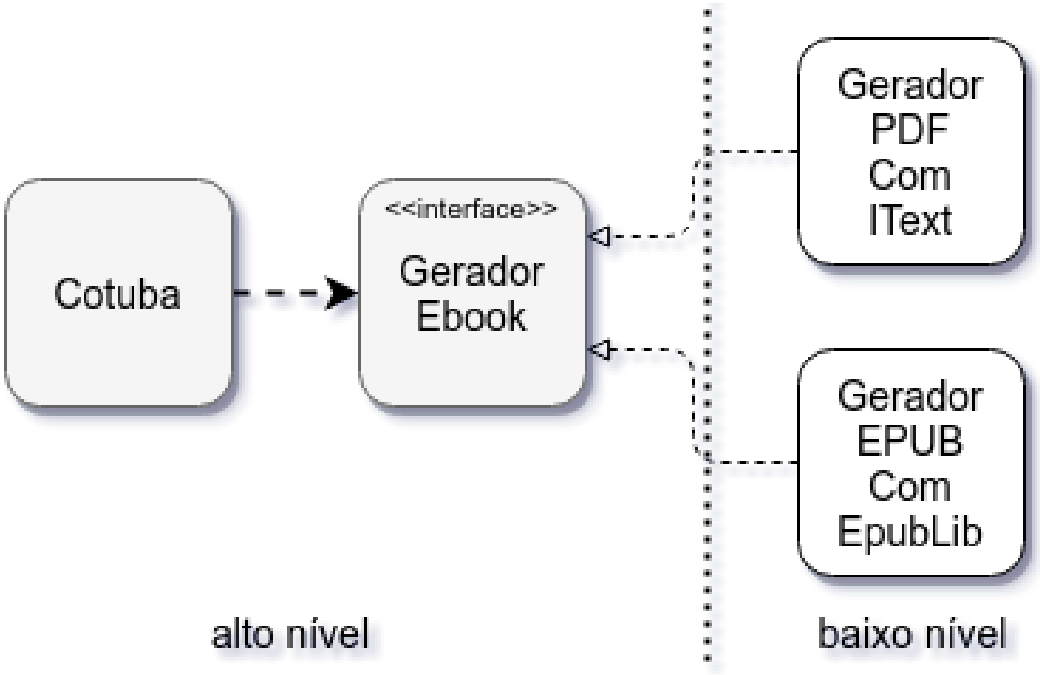

Figura 5.3: Abstração para geração de ebooks.

Uma abstração melhor para a geração de ebooks

Vamos criar uma abstração para a geração de ebooks, fazendo com que os geradores de PDF e EPUB sejam implementações. Primeiramente, vamos criar uma nova interface `GeradorEbook` no pacote `cotuba.application`:

```
// cotuba.application.GeradorEbook

package cotuba.application;

import cotuba.domain.Ebook;

public interface GeradorEbook {

  void gera(Ebook ebook);
```

```
}
```

Em seguida, vamos fazer com que a classe `GeradorPDFComIText` implemente a nova interface, removendo a anterior:

```
// cotuba.pdf.GeradorPDFComIText

// outros imports...
import cotuba.application.GeradorPDF;
import cotuba.application.GeradorEbook; // inserido

public class GeradorPDFComIText implements GeradorPDF {
public class GeradorPDFComIText implements GeradorEbook {

  // código omitido ...

}
```

O mesmo deve ser feito para a classe `GeradorEPUBComEpublib`:

```
// cotuba.epub.GeradorEPUBComEpublib

// outros imports...
import cotuba.application.GeradorEPUB;
import cotuba.application.GeradorEbook; // inserido

public class GeradorEPUBComEpublib implements GeradorEPUB {
public class GeradorEPUBComEpublib implements GeradorEbook {

  // código omitido ...

}
```

Vamos ignorar, por enquanto, os erros de compilação nas interfaces `GeradorPDF` e `GeradorEPUB`. Vamos corrigi-los em seguida!

Removendo interfaces desnecessárias

Já não precisamos das abstrações que tínhamos para a geração de PDFs e de EPUBs. Nesse momento da nossa refatoração, as

interfaces `GeradorPDF` e `GeradorEPUB` devem apresentar erros de compilação nos respectivos métodos `cria`. Essas interfaces não têm mais utilidade. Vamos removê-las sem dó!

```
cotuba.application.GeradorPDF
cotuba.application.GeradorEPUB
```

Agora o erro de acontecer na classe `Cotuba`, na criação das instâncias dos geradores. Vamos usar a interface `GeradorEbook`, instanciando os objetos apropriados, na classe `Cotuba`:

```
// cotuba.application.Cotuba

GeradorEbook gerador; // inserido

if ("pdf".equals(formato)) {

  GeradorPDF geradorPDF = GeradorPDF.cria();
  geradorPDF.gera(ebook);

  gerador = new GeradorPDFComIText(); // inserido

} else if ("epub".equals(formato)) {

  GeradorEPUB geradorEPUB = GeradorEPUB.cria();
  geradorEPUB.gera(ebook);

  gerador = new GeradorEPUBComEpublib(); // inserido

} else {
  throw new IllegalArgumentException(
        "Formato do ebook inválido: " + formato);
}

gerador.gera(ebook); // inserido
```

Pronto! Ao testarmos a geração de PDF e EPUB tudo deve funcionar!

Simplificando os nomes das implementações

Não há mais necessidade de revelarmos a tecnologia usada na implementação no nome da classe. Temos dois geradores de ebook. O que os diferencia é o formato gerado. Devemos adequar os nomes das implementações de `GeradorEbook`, simplificando-os. Para isso, vamos renomear as classes que implementam a interface `GeradorEbook` da seguinte maneira:

- de `GeradorEPUBComEpublib` para `GeradorEPUB`.
- de `GeradorPDFComIText` para `GeradorPDF`.

Ao definirmos a abstração de um gerador sem um formato específico, criamos uma maneira mais flexível de gerar ebooks. Para um novo formato de ebook, basta uma nova implementação de `GeradorEbook`.

A definição de uma abstração comum a várias implementações diferentes de um mesmo comportamento é descrita no **Strategy Pattern** do livro *Design Patterns* (GAMMA et al., 1994). Nos termos do livro, o Strategy define uma família de algoritmos intercambiáveis, encapsulados em um contrato comum.

Strategy ou Command?

No exemplo do começo do capítulo, da emissão de notas fiscais, a persistência no banco de dados e as notificações por e-mail e SMS foram "escondidas" atrás da interface `AcaoPosEmissao`. Já no exemplo do Cotuba, maneiras diferentes de gerar um ebook foram "escondidas" atrás da interface `GeradorEbook`.

Usamos o mesmo mecanismo nos dois casos: interfaces e suas implementações. Então, por que o primeiro caso é um exemplo do *Command Pattern* e o segundo, do *Strategy Pattern*? A principal diferença entre o *Command* e o *Strategy* não é o código resultante, mas o objetivo:

- No *Command*, queremos abstrair uma operação genérica, sem definir o que é feito.

- No *Strategy*, abstraímos um procedimento específico que faz coisas parecidas de jeitos diferentes.

5.5 Revisitando abstrações

Para respeitar o DIP, classificamos o código em alto e baixo nível e colocamos abstrações, comumente interfaces, nos pontos em que código de alto nível depende diretamente de código de baixo nível. Mas isso pode levar a uma proliferação desnecessária de abstrações.

Seguir o DIP à risca pode levar a códigos com muitas interfaces com apenas uma implementação. Isso já foi algo comum em alguns dos projetos em que o autor deste livro participou. Na maioria dos casos, eram limitações das próprias bibliotecas e frameworks. Mas, em alguns casos, era uma questão cultural do time de desenvolvimento, que tentava seguir boas práticas como se fossem mandamentos invioláveis, às vezes sem refletir sobre por que tais práticas foram adotadas.

Será que vale a pena manter uma abstração com apenas uma implementação? Não, mesmo que estejamos violando o DIP. Atender ao DIP de maneira burocrática faz com que tenhamos mais código para manter e não haja a desejada estabilidade nas dependências.

Cada abstração deve ser justificada por um aumento na flexibilidade do software. Mas, muitas vezes, precisamos dessa adaptabilidade apenas em pontos específicos do nosso código. E saber quais são esses pontos depende da estratégia e da natureza do software que está sendo desenvolvido. Para favorecer a flexibilidade, precisamos saber em quais partes do código devemos deixar as opções mais abertas. E talvez os pontos de flexibilidade mudem com o decorrer do projeto. Precisamos reavaliar as abstrações do nosso código continuamente.

Revisitando as abstrações do Cotuba

No caso do Cotuba, queremos ter a possibilidade de criar novos geradores de ebook rapidamente e modificar as implementações dos geradores que já existem. Por exemplo, pretendemos explorar a geração do PDF com LaTeX, em vez do iText usado atualmente. E desejamos criar geradores de AZW, que pode ser lido no leitor de ebooks Amazon Kindle, além de possíveis outros formatos. Porém, a geração de Markdown não é um ponto em que pensamos explorar. Podemos deixar os trechos de código relacionados a essa tarefa mais fechados.

Ao abordarmos o DIP, no capítulo anterior, classificamos a classe `Cotuba` como de alto nível e a classe responsável pela renderização de Markdown em HTML, a `RenderizadorMDParaHTMLComCommonMark`, como de baixo nível. Para respeitar o DIP, fazendo com que o código de alto nível não dependa de código de baixo nível, invertemos as dependências por meio da interface `RenderizadorMDParaHTML`, fazendo com que tanto o código de alto nível como o de baixo nível dependessem dessa abstração. Mas acabamos com uma interface com apenas uma implementação. Vamos remover a interface `RenderizadorMDParaHTML` ajustando a respectiva implementação:

```
import cotuba.application.RenderizadorMDParaHTML;

public class RenderizadorMDParaHTMLComCommonMark
   implements RenderizadorMDParaHTML {

  @Override
  public List<Capitulo> renderiza(Path diretorioDosMD) {
```

Em seguida, vamos instanciar a implementação `RenderizadorMDParaHTMLComCommonMark` diretamente na classe `Cotuba`:

```
// cotuba.application.Cotuba

RenderizadorMDParaHTML renderizador =
   RenderizadorMDParaHTML.cria();
var renderizador = new RenderizadorMDParaHTMLComCommonMark();
List<Capitulo> capitulos = renderizador.renderiza(diretorioDosMD);
```

A instanciação do `RenderizadorMDParaHTMLComCommonMark` é algo bem simples e, por isso, pode ser feita na própria classe `Cotuba`. Caso precisássemos de mais configurações e/ou dependências, poderíamos adotar um *factory* ou alguma outra abordagem para separar a criação do objeto de seu uso.

Já que removemos a abstração e temos apenas uma implementação do renderizador de Markdown, podemos renomear a classe, voltando ao seu nome original:

- de `RenderizadorMDParaHTMLComCommonMark` para `RenderizadorMDParaHTML`.

> É interessante notar que, em um projeto real, é comum que decisões de design sejam desfeitas, tanto pela correção de concepções erradas como por mudanças na direção do produto/projeto.

5.6 Uma Factory inteligente

Ao removermos as factories das antigas interfaces `GeradorPDF` e `GeradorEPUB`, que agora são nomes de classes, acabamos inserindo dependências indesejadas na classe `Cotuba`. Perceba pelos seguintes imports:

```
// cotuba.application.Cotuba

import cotuba.epub.GeradorEPUB;
import cotuba.pdf.GeradorPDF;
```

Essas dependências indesejadas são as classes concretas de geração de EPUBs e PDFs, respectivamente. São utilizadas na instanciação da nova abstração `GeradorEbook`:

```
// cotuba.application.Cotuba

if ("pdf".equals(formato)) {
  gerador = new GeradorPDF(); // implementação para PDF
} else if ("epub".equals(formato)) {
  gerador = new GeradorEPUB(); // implementação para EPUB
}
// restante do código...
```

Fizemos com que `Cotuba`, uma classe que havíamos classificado como de alto nível, dependesse das classes classificadas por nós como de baixo nível: `GeradorPDF` e `GeradorEPUB`.

Regras de negócio dependendo de mecanismos de entrega... Violamos o DIP! Como resolver?

Criando as instâncias das implementações fora do objeto que as usa. Ou seja, como uma *Factory*. Essa nova *Factory* será inteligente, já que terá uma lógica mais elaborada do que as que já

havíamos definido: dado um formato de ebook, será retornada uma instância da implementação correta.

Implementando uma Factory inteligente

Vamos definir uma *Factory* na interface `GeradorEbook` que, dado um formato, cria a instância apropriada.

Para isso, nosso passo inicial será adicionar um novo método estático na interface `GeradorEbook`, chamado `cria`, que recebe uma `String` no parâmetro `formato` e retorna uma instância de `GeradorEbook`:

```
// cotuba.application.GeradorEbook

public interface GeradorEbook {
  void gera(Ebook ebook);

  static GeradorEbook cria(String formato) { // inserido
  }

}
```

Em seguida, vamos mover a lógica da criação de instâncias de `GeradorEbook` da classe `Cotuba` para o novo método `cria`. Não podemos deixar de incluir o `return`:

```
// cotuba.application.GeradorEbook

GeradorEbook gerador;

if ("pdf".equals(formato)) {

  gerador = new GeradorPDF();

} else if ("epub".equals(formato)) {

  gerador = new GeradorEPUB();

} else {
```

```
  throw new IllegalArgumentException(
        "Formato do ebook inválido: " + formato);
}

return gerador; // ATENÇÃO para o return!
```

Devemos realizar os imports necessários para `GeradorPDF` e `GeradorEPUB`. Então, vamos usar a *Factory* de `GeradorEbook` no método `executa` da classe `Cotuba`:

```
// cotuba.application.Cotuba

GeradorEbook gerador = GeradorEbook.cria(formato); // modificado

gerador.gera(ebook);
```

Podemos limpar os imports desnecessários de `Cotuba`, referentes aos geradores de PDF e EPUB. A geração de ebooks deve funcionar! Nesse momento, as dependências da classe `Cotuba` estão algo como a seguinte imagem:

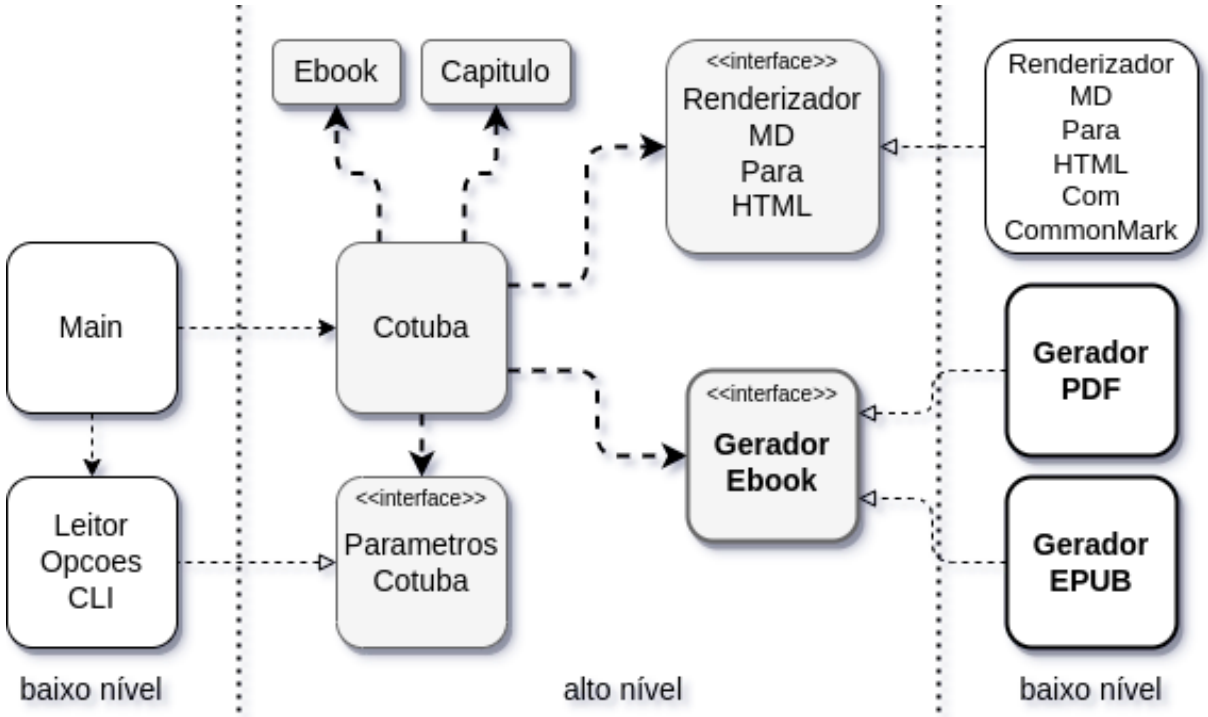

Figura 5.4: Dependências do Cotuba.

5.7 Das condicionais ao polimorfismo

À medida que avançamos nas refatorações, fomos caminhando na direção do polimorfismo e nos distanciando de condicionais na classe `Cotuba`. Uma boa abstração aliada a polimorfismo nos permite criar geradores de ebook em diferentes formatos sem modificar a classe que usa esses geradores. OCP na veia!

O livro *Refactoring* (FOWLER et al., 1999) cataloga a refatoração *Replace Conditional with Polymorphism* (Substituir Condicional por Polimorfismo). Ao mover cada ramificação da condicional para uma implementação diferente de uma abstração comum, usando polimorfismo, conseguimos simplificar o código da classe original.

No livro *Design Patterns* (GAMMA et al., 1994), os autores indicam que o Strategy pattern elimina condicionais: *"Quando diferentes comportamentos são agrupados em uma classe, é difícil evitar o uso de declarações condicionais para selecionar o comportamento correto. Encapsular o comportamento em classes de estratégia separadas elimina essas declarações condicionais."*

Um bom design OO vai ter poucas condicionais como `if-else` ou `switch-case`.

> *"Objetos têm um mecanismo fabuloso, mensagens polimórficas, que permitem expressar lógica condicional de maneira flexível mas clara.*
>
> *Ao trocar condicionais explícitos por mensagens polimórficas, comumente você consegue:*
>
> - *reduzir duplicação,*
> - *tornar seu código mais claro e*
> - *aumentar a flexibilidade.*
>
> *Tudo ao mesmo tempo."*
>
> Kent Beck, no livro *Refactoring* (FOWLER et al., 1999)

Até mais, condicionais!

Distanciamo-nos de condicionais em direção ao polimorfismo na classe `Cotuba`. Mas onde foram parar esses condicionais? No método `cria` da interface `GeradorEbook`, que é nossa implementação de uma *Factory* para geradores de ebook. Será que é possível livrar-nos totalmente de instruções *if-else* e *switch-case*?

Francesco Cirillo, desenvolvedor de software e criador da técnica Pomodoro de gerenciamento de tempo, iniciou em 2009 a provocadora *campanha Anti-IF* (CIRILLO, 2009): *"Muitos times querem ser ágeis, mas têm dificuldade em reduzir a complexidade de código. Vamos começar com algo concreto: saber como usar objetos de uma maneira que permita que os desenvolvedores eliminem os IFs ruins, aqueles que frequentemente comprometem a flexibilidade e a habilidade de evoluir do software."*

Como podemos eliminar os *if-else* ao máximo do código do Cotuba?

Mapeando formatos para geradores

Uma das maneiras de fazer isso é ter um `Map<String, GeradorEbook>` que, dado um determinado formato, contém a implementação de `GeradorEbook` correspondente. Definiríamos esse `Map` na interface `GeradorEbook`:

```
import java.util.HashMap;
import java.util.Map;

// outros imports...

public interface GeradorEbook {

  Map<String, GeradorEbook> GERADORES = new HashMap<>();

  // código omitido...
}
```

Apesar de parecer um atributo, `GERADORES` é uma constante, implicitamente `public static final`. Mas como inicializar os valores nesse `Map`? Não podemos definir blocos `static` em interfaces, muito menos construtores.

Podemos usar a sintaxe *double brace initialization*, que define um bloco de inicialização em uma subclasse anônima e é usado em algumas bibliotecas como a *jMock* (http://jmock.org). Veja:

```
public interface GeradorEbook {

  Map<String, GeradorEbook> GERADORES =
      new HashMap<String, GeradorEbook>() {{
    put("pdf", new GeradorPDF());
    put("epub", new GeradorEPUB());
  }};

  // código omitido...
}
```

> Perceba que não é possível usar o *diamond operator*, que removeria a necessidade de declaração dos tipos na instanciação do `HashMap`, ao usarmos inicialização com *double braces*.

Então, usaríamos o `Map` para obter os geradores a partir do formato, sem a necessidade de nenhuma condicional:

```
public interface GeradorEbook {

  // código omitido...

  static GeradorEbook cria(String formato) {
    GeradorEbook gerador = GERADORES.get(formato);

    if (gerador == null) {
      throw new IllegalArgumentException(
            "Formato do ebook inválido: " + formato);
    }

    return gerador;
  }
```

Guilherme Silveira explora uma implementação parecida, usando mapas, e uma outra usando a Reflection API do Java no artigo *Como não aprender orientação a objetos: o excesso de ifs* (SILVEIRA, 2011).

Usando o poder das enums

Até agora, os formatos PDF e EPUB estão representados no código como `String`. Poderíamos defini-los em uma enum. Como é um conceito do domínio, podemos colocá-la no pacote `cotuba.domain`:

```
package cotuba.domain;

public enum FormatoEbook {
```

```
  PDF, EPUB;

}
```

Enums são tipos bastante poderosos em Java: podem ter atributos, construtores (implicitamente privados) e métodos públicos ou privados. Podemos usar esse poder das enums para evitar condicionais:

```
import cotuba.application.GeradorEbook;
import cotuba.epub.GeradorEPUB;
import cotuba.pdf.GeradorPDF;

public enum FormatoEbook {

  PDF(new GeradorPDF()),
  EPUB(new GeradorEPUB());

  private GeradorEbook gerador;

  FormatoEbook(GeradorEbook gerador) {
    this.gerador = gerador;
  }

  public GeradorEbook getGerador() {
    return gerador;
  }

}
```

Na classe `Ebook`, devemos mudar o tipo do atributo `formato` para `FormatoEbook` e corrigir os *getters* e os *setters*:

```
// cotuba.domain.Ebook

public class Ebook {

  private String formato;
  private FormatoEbook formato;

  private Path arquivoDeSaida;
```

```
  private List<Capitulo> capitulos;

  public String getFormato() {
  public FormatoEbook getFormato() {
    return formato;
  }

  public void setFormato(String formato) {
  public void setFormato(FormatoEbook formato) {
    this.formato = formato;
  }

  // restante do código...

}
```

Deve acontecer um erro de compilação na classe `Cotuba`. Para corrigi-lo, devemos usar a nova enum como tipo da variável `formato`:

```
// cotuba.application.Cotuba

import cotuba.domain.FormatoEbook; // inserido

public class Cotuba {

  public void executa(ParametrosCotuba parametros) {

    String formato = parametros.getFormato();
    FormatoEbook formato = parametros.getFormato();

    // código omitido...

  }

}
```

Devem acontecer novos erros de compilação. Vamos arrumá-los aos poucos. Na interface `ParametrosCotuba`, precisamos corrigir o tipo de retorno do método `getFormato`:

```
// cotuba.application.ParametrosCotuba

import cotuba.domain.FormatoEbook; // inserido

public interface ParametrosCotuba {

  String getFormato();
  FormatoEbook getFormato();

  // outros métodos...

}
```

Em `LeitorOpcoesCLI`, devemos mudar o tipo do atributo `formato` para `FormatoEbook` e corrigir o *getter* e o construtor.

```
// cotuba.cli.LeitorOpcoesCLI

import cotuba.domain.FormatoEbook; // inserido

class LeitorOpcoesCLI implements ParametrosCotuba {

  private String formato;
  private FormatoEbook formato;

  // outros parâmetros...

  public LeitorOpcoesCLI(String[] args) {

    // código omitido...

    formato = nomeDoFormatoDoEbook.toLowerCase();
    formato = FormatoEbook.valueOf(
          nomeDoFormatoDoEbook.toUpperCase());

    // código omitido...

    formato = "pdf";
    formato = FormatoEbook.PDF;

    // código omitido...
```

```
    arquivoDeSaida = Paths.get("book." + formato.toLowerCase());
    arquivoDeSaida = Paths.get("book." +
        formato.name().toLowerCase());

    // restante do código...

  }

  @Override
  public String getFormato() {
  public FormatoEbook getFormato() { // modificado
    return formato;
  }

  // outros getters...

}
```

Na interface `GeradorEbook`, devemos corrigir o método `cria`, para que receba um `FormatoEbook`. Em seguida, devemos fazer com que a instância do gerador seja obtida da própria enum:

```
// cotuba.application.GeradorEbook

import cotuba.domain.FormatoEbook; // inserido

public interface GeradorEbook {

  // código omitido...

  static GeradorEbook cria(String formato) {
  static GeradorEbook cria(FormatoEbook formato) {

    GeradorEbook gerador;
    if ("pdf".equals(formato)) {
      gerador = new GeradorPDF();
    } else if ("epub".equals(formato)) {
      gerador = new GeradorEPUB();
    } else {
      throw new IllegalArgumentException(
```

```
      "Formato do ebook inválido: " + formato);
    }

    return formato.getGerador();

  }

}
```

Diversas linhas de código removidas mantendo a mesma funcionalidade! Deve ser inserido o import a enum `FormatoEbook` e removidos os referentes às implementações dos geradores de ebooks. O código deve compilar sem erros. Ao testarmos a geração de PDF e EPUB, tudo deve funcionar!

5.8 Pondo a flexibilidade do Cotuba à prova

Será que a criação de novos geradores de ebook está realmente flexível? Podemos validar isso criando um gerador em um novo formato.

Vamos dizer que desejamos gerar um site com o conteúdo do nosso livro. Para isso, definiríamos HTML como novo formato do ebook. Os HTMLs renderizados a partir dos arquivos Markdown de cada capítulo seriam salvos em um diretório. Teríamos que:

- criar uma nova implementação de `GeradorEbook`, que vamos chamar de `GeradorHTML`;
- modificar a enum `FormatoEbook`, adicionando HTML como um formato e passando uma instância de `GeradorHTML`.

Vamos começar criando a classe geradora de HTML. Essa classe:

- verifica se o arquivo de saída já existe e, se existir, o deleta. No caso de um diretório, todo o conteúdo interno também deve ser deletado;
- cria o novo diretório de saída;
- para cada capítulo, cria um novo arquivo `.html` com o conteúdo renderizado a partir do Markdown, apenas com letras minúsculas no nome e o número do capítulo como prefixo.

O código da classe `GeradorHTML`, que deve ser criada em um novo pacote `cotuba.html`, ficaria algo como:

```
// cotuba.html.GeradorHTML

package cotuba.html;

import cotuba.application.GeradorEbook;
import cotuba.domain.Capitulo;
```

```
import cotuba.domain.Ebook;

import java.io.IOException;
import java.nio.charset.StandardCharsets;
import java.nio.file.Files;
import java.nio.file.Path;
import java.text.Normalizer;

public class GeradorHTML implements GeradorEbook {

  @Override
  public void gera(Ebook ebook) {
    Path arquivoDeSaida = ebook.getArquivoDeSaida();
    try {
      Path diretorioDoHTML =
          Files.createDirectory(arquivoDeSaida);
      int i = 1;
      for (Capitulo capitulo : ebook.getCapitulos()) {
        String nomeDoArquivoHTMLDoCapitulo =
            obtemNomeDoArquivoHTMLDoCapitulo(i, capitulo);
        Path arquivoHTMLDoCapitulo =
diretorioDoHTML.resolve(nomeDoArquivoHTMLDoCapitulo);
        String html = """
            <!DOCTYPE html>
            <html lang="pt-BR">
              <head>
                <meta charset="UTF-8">
                <title>%s</title>
              </head>
              <body>
                %s
              </body>
            </html>
            """.formatted(capitulo.getTitulo(),
                capitulo.getConteudoHTML());
        Files.writeString(arquivoHTMLDoCapitulo, html,
            StandardCharsets.UTF_8);
        i++;
      }
    } catch (IOException ex) {
      throw new IllegalStateException("Erro ao criar HTML: "
```

```
          + arquivoDeSaida.toAbsolutePath(), ex);
    }
  }

  private String obtemNomeDoArquivoHTMLDoCapitulo(int i,
      Capitulo capitulo) {
    String nomeArquivoHTMLCapitulo = i + "-"
      + removeAcentos(capitulo.getTitulo().toLowerCase())
          .replaceAll("[^\\w]", "")
      + ".html";
    return nomeArquivoHTMLCapitulo;
  }

  // Será que remover acentos é responsabilidade do GeradorHTML?
  private String removeAcentos(String texto) {
    return Normalizer.normalize(texto, Normalizer.Form.NFD)
      .replaceAll("[^\\p{ASCII}]", "");
  }

}
```

A classe `Normalizer`, do pacote `java.text`, é utilizada em um método auxiliar para remover acentos do título do capítulo, afim de criar um nome de arquivo.

Então, precisamos adicionar o formato HTML e uma instância da nova classe na enum `FormatoEbook`:

```
// cotuba.domain.FormatoEbook

import cotuba.html.GeradorHTML; // inserido

public enum FormatoEbook {

  PDF(new GeradorPDF()),
  EPUB(new GeradorEPUB()),
  HTML(new GeradorHTML()); // inserido

  // restante do código...

}
```

Pronto! Não precisamos alterar mais nenhum ponto do nosso sistema. Estamos abertos para extensão de geradores de ebook e fechados para modificação do resto do código. O ponto de flexibilidade que projetamos foi validado com sucesso!

Ao testarmos passando HTML como formato, por meio da opção `-f html`, teremos um site com o conteúdo HTML do nosso livro!

Para saber mais: OCP e Spring

Com um framework de Dependency Injection como o Spring, podemos implementar o OCP de maneira muito elegante e flexível. Vamos considerar que temos a interface `GeradorEbook` com três implementações: a classe `GeradorPDF`, a classe `GeradorEPUB` e a classe `GeradorHTML`. Todas essas implementações estariam anotadas com `@Component`, conforme mencionado no capítulo sobre o DIP.

Em um projeto cujas dependências são gerenciadas pelo Spring, ao injetarmos listas de uma determinada interface, obtemos todos os componentes gerenciados pelo framework que são implementações da interface.

Podemos modificar a classe `Cotuba`, que transformamos em um `@Component` do Spring no capítulo anterior, para que receba uma lista com todos os geradores de ebook em vez de cada um separadamente:

```
// cotuba.application.Cotuba

@Component
public class Cotuba {

  private final List<GeradorEbook> geradoresEbook; // modificado
  private final RenderizadorMDParaHTML renderizador;

  public Cotuba(List<GeradorEbook> geradoresEbook,
                RenderizadorMDParaHTML renderizador) {
      this.geradoresEbook = geradoresEbook; // modificado
      this.renderizador = renderizador;
```

```
  }

  // restante do código...

}
```

No método `executa` da classe `Cotuba`, devemos utilizar a lista:

```
// cotuba.application.Cotuba

@Component
public class Cotuba {

  // código omitido...

  public void executa(ParametrosCotuba parametros) {

    // código omitido...

    // modificado
    for (GeradorEbook geradorEbook : geradoresEbook) {
      geradorEbook.gera(ebook);
    }

}
```

Opa! Acabamos de inserir um bug: sempre serão gerados ebooks em todos os formatos. Precisamos, de alguma forma, descobrir qual o gerador de ebook apropriado para o formato escolhido pelo usuário.

Existem algumas maneiras de implementar essa tarefa. Uma abordagem é definir a enum `FormatoEbook` e associar seus valores às classes que implementam a geração de ebooks para cada formato. Para o formato `PDF`, teríamos a classe `GeradorPDF.class` e, para o `EPUB`, o `GeradorEPUB.class`. A partir dessas classes, usaríamos o `ApplicationContext` para obter instâncias gerenciadas pelo Spring, com todas as suas dependências.

Outra abordagem é definir na própria interface um método que indica se o formato é suportado ou não por uma implementação específica. Talvez seja a abordagem mais comum em projetos Spring e é a que adotaremos!

Escolhida a abordagem, vamos definir um método `accept` na interface `GeradorEbook`, que recebe um formato de ebook e retorna um `boolean`, indicando se o formato é aceito pelo gerador:

```
// cotuba.application.GeradorEbook

public interface GeradorEbook {

  void gera(Ebook ebook);

  boolean accept(FormatoEbook formato); // inserido

}
```

Vamos implementar o novo método na classe `GeradorPDF`:

```
// cotuba.application.GeradorPDF

import cotuba.domain.FormatoEbook;

@Component
public class GeradorPDF implements GeradorEbook {

  // código omitido...

  @Override
  public boolean accept(FormatoEbook formato) {
    return FormatoEbook.PDF.equals(formato);
  }

}
```

O mesmo deve ser feito para as classes `GeradorEPUB` e `GeradorHTML`, definindo os formatos `EPUB` e `HTML`, respectivamente.

A enum `FormatoEbook` pode ser simplificada, removendo o atributo que referenciava os geradores de ebook:

```
// cotuba.domain.FormatoEbook

public enum FormatoEbook {
  PDF, EPUB, HTML;
}
```

Nesse momento, vamos modificar o método `executa` da classe `Cotuba`, usando uma pitada de Streams e Lambdas:

```
// cotuba.application.Cotuba

@Component
public class Cotuba {

  // código omitido...

  public void executa(ParametrosCotuba parametros) {

    // código omitido...

    GeradorEbook geradorEbook = geradoresEbook.stream()
      .filter(gerador -> gerador.accept(formato))
      .findAny()
      .orElseThrow(() -> new IllegalArgumentException(
            "Formato do ebook inválido: " + formato));

    geradorEbook.gera(ebook);

  }

}
```

Pronto! Usando o poder do Spring, para definir um novo gerador de ebook bastaria adicionar uma nova classe que implementa a interface `GeradorEbook`. Ampliaríamos as funcionalidades do software apenas adicionando código, com o mínimo de alterações possível. Esse é o verdadeiro intuito do OCP!

O código anterior pode ser encontrado em: https://github.com/alexandreaquiles/solid-na-pratica/tree/cap5-extra-open-closed-e-spring

Para saber mais: o OCP de Bertrand Meyer

No livro *Object-Oriented Software Construction* (MEYER, 1988), o pioneiro da Orientação a Objetos Bertrand Meyer cunha o OCP ao listar critérios, regras e princípios para modularidade.

Os critérios de Meyer para código modular são: decomponibilidade, composibilidade, compreensibilidade, continuidade e proteção.

As regras da modularidade do autor são: mapeamento direto, poucas interfaces, interfaces pequenas, interfaces explícitas e ocultação de informação (*information hiding*).

Os princípios de Meyer para atingir modularidade são: o princípio das unidades modulares linguísticas, o princípio da autodocumentação, o princípio do acesso uniforme, o princípio da escolha única e, finalmente, o **princípio aberto/fechado** (o OCP).

Meyer diz que *"módulos devem ser tanto abertos como fechados."* Módulos devem ser abertos porque devem ainda estar disponíveis para extensão, por exemplo, expandindo as operações ou adicionando atributos em suas estruturas de dados.

Módulos são considerados fechados se estiverem disponíveis para uso por outros módulos: devem ter uma descrição estável e bem definida; devem ser compilados, armazenados em uma biblioteca e disponibilizados para quem os usa; devem ter sua interface publicada (aqui no sentido de API) e devem ser adicionados ao repositório oficial do projeto.

Para Meyer, a abertura é uma preocupação natural já que é quase impossível prever todos os elementos necessários para um módulo e, portanto, é desejável manter o máximo de flexibilidade para abarcar futuras alterações e extensões. Mas é igualmente

necessário entregar os módulos para outros desenvolvedores e clientes e, para isso, precisamos fechá-los.

Segundo o autor, poderíamos implementar o OCP copiando e colando código em outros módulos, o que levaria a uma explosão de variantes do módulo original, muitas delas muito semelhantes entre si, embora nunca exatamente idênticas. Então, o autor diz algo que lembra o acrônimo D.R.Y. *(Don't Repeat Yourself)*: *"Melhor evitar a redundância do que gerenciá-la."*

Poderíamos também ceder ao impulso de modificar o código para adicionar novos casos, o que faria com que o código ficasse poluído por condicionais. Meyer indica que OO possui um mecanismo elegante para implementar o OCP: herança. OCP em uma linguagem OO permite atender às variantes sem afetar a consistência da versão original. O autor conclui com algumas ressalvas em relação ao OCP:

- Se você tiver controle sobre o software original e puder reescrevê-lo para que atenda às necessidades de vários tipos de cliente sem nenhuma complicação extra, deveria fazê-lo.
- OCP não deve ser uma maneira de abordar falhas de modelagem, muito menos uma alterativa para contornar bugs. O módulo original deve ser saudável e seus defeitos devem ser corrigidos.

5.9 Contraponto: críticas ao OCP

Dan North, no artigo *CUPID - the back story* (NORTH, 2021), liga o OCP a um contexto já ultrapassado, em que modificar software era caro e arriscado. Com técnicas modernas como refatoração e TDD, o código é muito mais maleável. Dessa forma, código deixa de ser um ativo a ser preservado e passa a ser um custo a ser minimizado. De acordo com North, nesse contexto atual, escrever código simples e mais específico passaria a fazer mais sentido.

David Copeland diz, no livro *SOLID is not solid* (COPELAND, 2019), que o OCP de Meyer estaria relacionado com um contexto em que a compilação de código era muito demorada; software ainda não era distribuído pela internet, mas por cópias físicas como CD-ROM; e sistemas eram construídos em torno de bibliotecas compartilhadas. Nesse contexto, recompilar e redistribuir o sistema como um todo era bastante oneroso tanto em tempo como em dinheiro. Por isso, seria interessante criar softwares extensíveis, mesmo que com código mais difícil de entender.

Copeland critica também a descrição do OCP por Uncle Bob, que daria a entender que todas as classes deveriam ser altamente extensíveis. Copeland conclui: *"(...) não adicione toneladas de flexibilidade da primeira vez só porque você pode precisar mudar as coisas depois."*

Kevlin Henney, na palestra *The SOLID Design Principles Deconstructed* (HENNEY, 2013), resgata o texto do livro *Object-Oriented Software Construction* (MEYER, 1988) de Bertrand Meyer, que é citado por Uncle Bob como a inspiração do OCP. Henney mostra que a motivação por trás do OCP original era reúso e que estava ligado a versionamento de módulos. Diz ainda que quando o livro foi escrito, na década de 80, o uso de sistemas de controle de

versão e gerenciamento de dependências ainda não era comum. Henney tece um paralelo com o conceito de *published interfaces* (interfaces publicadas) do livro *Refactoring* (FOWLER et al., 1999) de Martin Fowler: classes ou interfaces que são usadas fora da base de código que as define.

Ted Kaminski, em seu artigo *Deconstructing SOLID design principles* (KAMINSKI, 2019), argumenta que extensibilidade deveria ser um objetivo apenas nas *bordas* do sistema: as interfaces expostas publicamente para desenvolvedores de terceiros. Código que não está exposto nas bordas do sistema é barato de modificar e não precisa ser extensível: *"(...) você pode até dizer que deveríamos ser abertos para modificação."*

5.10 O que aprendemos?

Neste capítulo, fizemos várias alterações na direção de boas abstrações e de um código sem *if-else*. Agora, para definir um novo formato de ebook, basta criarmos mais uma implementação de `GeradorEbook` e mais um valor na enum `FormatoEbook`. Não precisamos alterar o código da classe `Cotuba`, nem de outras classes! Ao seguirmos o OCP, fizemos um código que pode ser estendido em suas funcionalidades sem ser modificado.

No próximo capítulo, vamos estudar uma maneira extrema do OCP: os *plugins*, que permitem que código cuja existência desconhecemos seja usado para estender as funcionalidades de um sistema.

O código deste capítulo pode ser encontrado em: https://github.com/alexandreaquiles/solid-na-pratica/tree/cap5-open-closed-principle

Figura 5.5: Vídeo do capítulo

CAPÍTULO 6

OCP potencializado: plugins

Fizemos o lançamento do Cotuba e o projeto foi muito bem-sucedido! E, com o sucesso, vieram diversas demandas de novas funcionalidades por diferentes empresas.

Algumas das empresas querem manipular o HTML renderizado a partir do Markdown, antes de gerar os ebooks. Entre as solicitações de modificação do HTML estão:

- estilizar os capítulos aplicando CSS;
- adicionar cabeçalhos e rodapés;
- adicionar propagandas no início de cada capítulo;
- incluir legendas nas imagens.

Outras empresas solicitaram modificações no ebook gerado como:

- calcular estatísticas para o ebook, como as palavras mais comuns;
- incluir uma página de *copyright* como primeira página do ebook;
- incluir sumário nas páginas iniciais do ebook, quando estiver no formato PDF.

Nosso time é enxuto e, nesse momento do projeto, estamos focados especificamente em melhorar o PDF gerado. É inviável criarmos funcionalidades no próprio Cotuba para atender a cada uma das demandas requisitadas.

Como podemos oferecer flexibilidade o bastante para que as equipes de desenvolvimento das próprias empresas consigam implementar customizações?

6.1 Um plugin no Cotuba

Uma ideia seria pedir que as empresas desenvolvessem suas soluções e nos enviassem o código já pronto, talvez por meio de *Pull Requests* no GitHub. Mas o esforço de coordenação seria muito grande. Teríamos que analisar cada linha de código enviada, verificando se há novos bugs, problemas de segurança e se a qualidade do código é boa o suficiente para ser incorporada ao código do Cotuba. Uma vez que aceitássemos o código, a manutenção seria por nossa conta. Além disso, uma implementação enviada por uma empresa valeria para todas as outras.

Seria melhor uma abordagem em que as próprias empresas fizessem suas implementações de maneira totalmente independente, sem afetar umas às outras e sem demandar revisão nem manutenção por parte de nossa equipe. Devemos criar uma maneira de permitir que classes de outras empresas sejam aplicadas durante a execução da aplicação, sem a necessidade de termos o código dessas classes durante o desenvolvimento do Cotuba. Para isso, vamos criar um ponto de extensão para o nosso código: um **plugin**!

Definiremos uma nova abstração, por meio da interface `Plugin`. Nessa interface, vamos definir dois métodos, alinhados com as demandas das empresas que foram listadas anteriormente:

- um método que poderá ser utilizado para customizações a serem aplicadas após a renderização do HTML dos capítulos.
- um método para customizações a serem feitas depois da geração do ebook.

Esse tipo de plugin que está associado a partes do ciclo de vida de uma aplicação é comumente chamado de *hook* (gancho, inglês). A

ideia é definir um hook para a renderização do HTML e um hook para a geração do ebook.

No pacote `cotuba.plugin`, vamos criar a interface `Plugin`, com:

- um método `aposRenderizacao` para o hook de renderização, que recebe um HTML de um capítulo e retorna uma `String` com o HTML modificado;
- um método `aposGeracao` para o hook de geração, que recebe o ebook gerado.

```
// cotuba.plugin.Plugin

package cotuba.plugin;

public interface Plugin {

  String aposRenderizacao(String html);

  void aposGeracao(Ebook ebook);

}
```

Agora, projetos de terceiros poderão utilizar a interface `Plugin` para implementar as personalizações desejadas.

6.2 Aplicando temas CSS nos capítulos

Digamos que um dos nossos clientes é a empresa Paradizo, que quer definir seu próprio estilo no ebook gerado. Para realizar essa tarefa, a equipe de desenvolvimento da Paradizo tem como objetivo a inserção de um código CSS nos HTMLs de cada capítulo. A equipe da Paradizo deve começar criando um projeto Maven chamado `tema-paradizo` com as seguintes configurações:

- *Group Id*: `br.com.paradizo`
- *Artifact Id*: `tema-paradizo`

Em seguida, a equipe da Paradizo também deve configurar:

- a *Version* como `0.0.1-SNAPSHOT`
- o *Packaging* como `jar`
- a codificação de caracteres como UTF-8
- o Java 17 tanto para o código-fonte como para o código compilado nos `.class`

O `pom.xml` resultante deve ser parecido com o seguinte:

```
<!-- tema-paradizo/pom.xml -->

<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>br.com.paradizo</groupId>
  <artifactId>tema-paradizo</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <packaging>jar</packaging>

  <properties>
```

```
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <maven.compiler.source>17</maven.compiler.source>
    <maven.compiler.target>17</maven.compiler.target>
  </properties>

</project>
```

O estilo customizado pensado pela equipe da Paradizo tem: uma borda tracejada abaixo do título do capítulo; uma borda sólida à esquerda e abaixo dos títulos das seções; e uma borda sólida em torno das citações. Para isso, a equipe da Paradizo deve definir um arquivo `tema.css` no diretório `src/main/resources` do projeto `tema-paradizo`:

```
/* tema-paradizo/src/main/resources/tema.css */

h1 { /* título do capítulo */
  border-bottom: 1px dashed black;
  font-size: 3em;
}

h2 { /* título das seções */
  border-left: 1px solid black;
  padding-left: 5px;
  border-bottom: 1px solid black;
}

blockquote { /* citações */
  border: 1px solid black;
  padding: 5px;
}
```

A equipe da Paradizo também deve criar uma classe `TemaParadizo` no pacote `br.com.paradizo.tema`, que terá a responsabilidade de aplicar o CSS no HTML de cada capítulo.

```
// br.com.paradizo.tema.TemaParadizo

package br.com.paradizo.tema;
```

```
public class TemaParadizo {

}
```

Para obter o `tema.css` a partir da classe `TemaParadizo`, a equipe deve definir uma classe utilitária `FileUtils`, no mesmo pacote `br.com.paradizo.tema`. Essa classe ajuda a obter o conteúdo de arquivos de texto que podem estar dentro ou fora de JARs.

> Você pode encontrar o código a seguir na URL: http://bit.ly/fj38-file-utils

```
// br.com.paradizo.tema.FileUtils

package br.com.paradizo.tema;

public class FileUtils {

  public static String getResourceContents(String resource) {
    try {
      Path resourcePath = getResourceAsPath(resource);
      return getPathContents(resourcePath);
    } catch(URISyntaxException | IOException ex) {
      throw new IllegalStateException(ex);
    }
  }

  private static Path getResourceAsPath(String resource)
      throws URISyntaxException, IOException {
    URI uri = FileUtils.class.getResource(resource).toURI();
    if (isResourceInJar(uri)) {
      return getResourceFromJar(uri);
    } else {
      return Paths.get(uri);
    }
  }

  private static boolean isResourceInJar(URI uri) {
    return uri.getScheme().equals("jar");
```

```
  }

  private static Path getResourceFromJar(URI fullURI)
      throws IOException {
    String[] uriParts = fullURI.toString().split("!");
    URI jarURI = URI.create(uriParts[0]);
    FileSystem fs;
    try {
      fs = FileSystems.newFileSystem(jarURI,
          Collections.<String, String>emptyMap());
    } catch (FileSystemAlreadyExistsException ex) {
      fs = FileSystems.getFileSystem(jarURI);
    }
    String resourceURI = uriParts[1];
    return fs.getPath(resourceURI);
  }

  private static String getPathContents(Path path)
      throws IOException {
    return Files.readString(path);
  }

}
```

Devem ser feitos imports de classes dos pacotes `java.io`, `java.net`, `java.nio.file` e `java.util`. Na classe `TemaParadizo`, a classe `FileUtils` deve ser usada pelo time da Paradizo para obter o conteúdo do arquivo `tema.css`:

```
// br.com.paradizo.tema.TemaParadizo

public class TemaParadizo {

  private String cssDoTema() {
    return FileUtils.getResourceContents("/tema.css");
  }

}
```

Em seguida, a equipe da Paradizo deve definir um método `aplicaTema`, que recebe uma `String` com um HTML e retorna um

novo HTML com o tema CSS aplicado.

```
// br.com.paradizo.tema.TemaParadizo

public class TemaParadizo {

  private String aplicaTema(String html) {
    // implementação aqui...
  }

  // código omitido...

}
```

Por enquanto, o código anterior apresentará erros de compilação.

Com o objetivo de aplicar o CSS no HTML, a equipe da Paradizo utilizará o Jsoup, uma biblioteca Java de manipulação de HTML que tem uma API que lembra a do JQuery. Para usar o Jsoup, a equipe deve usar o Maven para declarar a biblioteca como dependência:

```
<!-- tema-paradizo/pom.xml -->

<dependency>
  <groupId>org.jsoup</groupId>
  <artifactId>jsoup</artifactId>
  <version>1.11.2</version>
</dependency>
```

Para que o CSS do tema seja aplicado ao HTML recebido no método `aplicaTema`, o time da Paradizo deve usar o Jsoup para adicionar uma tag `style` no final do `head` do HTML:

```
// br.com.paradizo.tema.TemaParadizo

import org.jsoup.Jsoup;  // inserido
import org.jsoup.nodes.Document;  // inserido

public class TemaParadizo {

  private String aplicaTema(String html) {
```

```
    Document document = Jsoup.parse(html);

    String css = cssDoTema();

    document.select("head")
      .append("<style> " + css + " </style>");

    return document.html();

  }

  // código omitido...

}
```

Ótimo! O time de desenvolvimento da Paradizo já tem, na classe `TemaParadizo`, um código preparado para aplicar um tema CSS em um HTML de um capítulo. Mas como o pessoal da Paradizo vai ligar a classe `TemaParadizo` com a interface `Plugin` do Cotuba?

Para saber mais: a biblioteca Jsoup

Jsoup é uma biblioteca feita em Java que provê uma API baseada no jQuery para manipular HTML. Considere o HTML a seguir:

```
<div class="curso">
  <h2 class="curso__titulo">Curso Design de código SOLID em Java</h2>
  <p class="curso__info"><span>20</span> horas/aula</p>
</div>
```

Podemos usar o Jsoup para extrair texto e até mudar o HTML:

```
String html = //...

Document doc = Jsoup.parse(html);

Elements info = doc.select(".curso__info"); // <p>

String texto = info.text(); // "20 horas/aula"
```

```
// adiciona link depois do <p>
doc.select(".curso").append("<a href=\"#turmas\">Turmas</a>");

String novoHtml = doc.html(); // HTML atualizado
```

Implementando o plugin de tema

Para fazer com que a classe `TemaParadizo` implemente a interface `Plugin` do Cotuba, as pessoas da Paradizo precisam declarar o projeto `cotuba-cli` como dependência do projeto `tema-paradizo`:

```
<!-- tema-paradizo/pom.xml -->

<dependencies>

  <dependency>
    <groupId>cotuba</groupId>
    <artifactId>cotuba-cli</artifactId>
    <version>0.0.1-SNAPSHOT</version>
  </dependency>

</dependencies>
```

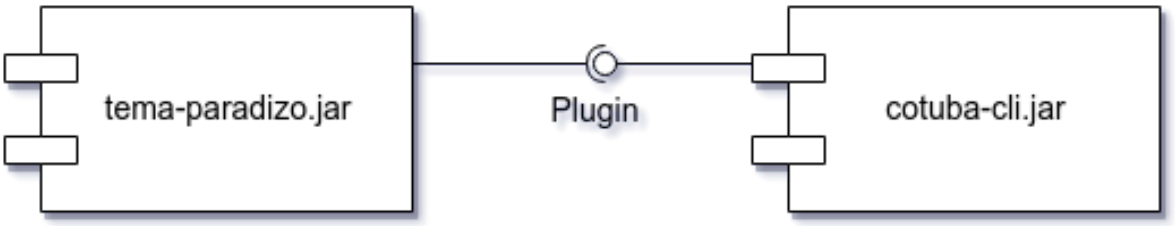

Figura 6.1: Tema Paradizo implementa interface do Cotuba

Com o `cotuba-cli` declarado como dependência, o time da Paradizo pode implementar a interface `Plugin`, definindo um comportamento para o hook de renderização. Para isso, o método `aposRenderizacao` deve aplicar o tema no HTML recebido:

```
// br.com.paradizo.tema.TemaParadizo

import cotuba.plugin.Plugin; // inserido
```

```
public class TemaParadizo implements Plugin { // modificado

  // código omitido...

  // inserido
  @Override
  public String aposRenderizacao(String html) {

    String htmlComTema = aplicaTema(html);
    return htmlComTema;

  }

  // inserido
  @Override
  public void aposGeracao(Ebook ebook) {
    // nada aqui...
  }


}
```

Também é necessário definir uma implementação para o método `aposGeracao` de `Plugin`. O que fazer? É possível simplesmente não fazer nada nesse método, por enquanto. Em capítulos posteriores, discutiremos se essa é uma boa decisão.

Tentando executar o plugin de tema

Uma questão importante é que precisamos de uma maneira de disponibilizar o JAR do `cotuba-cli` para o time da Paradizo, para que seja possível ter acesso à interface `Plugin`.

Como o Cotuba é um projeto open-source, a equipe da Paradizo pode baixar o projeto e fazer o build na própria máquina. Nos primeiros capítulos, para efetuar o build do Cotuba, utilizamos o comando:

```
# cotuba-cli

mvn clean package
```

Esse comando limpa possíveis artefatos anteriores, baixa as dependências, compila o código e, depois de diversas outras tarefas, gera os artefatos que, no caso do Cotuba, são o JAR do Cotuba e um ZIP.

```
[INFO] --- maven-jar-plugin:2.4:jar (default-jar) @ cotuba-cli ---
[INFO] Building jar: .../cotuba/target/cotuba-cli-0.0.1-SNAPSHOT.jar
[INFO]
[INFO] --- maven-assembly-plugin:3.1.0:single (make-assembly) @ cotuba-cli
---
[INFO] Reading assembly descriptor: src/assembly/distribution.xml
...
[INFO] Building zip: .../cotuba/target/cotuba-cli-0.0.1-SNAPSHOT-
distribution.zip
```

O ZIP gerado contém, além do JAR do Cotuba, os JARs de todas as dependências. A equipe da Paradizo pode aproveitar os artefatos gerados pelo `mvn package` e descompactar o `.zip` para o Desktop com o comando:

```
# cotuba-cli

unzip -o target/cotuba-*-distribution.zip -d ~/Desktop
```

Nesse momento, teremos no Desktop o arquivo `cotuba.sh` e, no diretório `libs`, o JAR do Cotuba junto aos JARs de todas as dependências:

```
Desktop
├── cotuba-cli-0.0.1-SNAPSHOT-distribution.zip
├── cotuba.sh
└── libs
    ├── commonmark-0.11.0.jar
    ├── commons-cli-1.4.jar
    ├── cotuba-cli-0.0.1-SNAPSHOT.jar  <-- JAR do Cotuba
    ├── epublib-core-3.1.jar
    ├── html2pdf-2.0.0.jar
```

```
├── io-7.1.0.jar
├── kernel-7.1.0.jar
├── kxml2-2.5.1-20180703.211101-1-jar-with-dependencies.jar
├── layout-7.1.0.jar
├── slf4j-api-1.6.1.jar
└── slf4j-simple-1.6.1.jar
```

O artefato que interessa ao time de desenvolvimento da Paradizo é o `cotuba-cli-0.0.1-SNAPSHOT.jar`. Porém, como fazer com que esse JAR do Cotuba seja considerado pelo Maven como uma dependência? Podemos disponibilizá-lo junto a alguns arquivos XML de configuração, no repositório local do Maven que, por padrão, fica no diretório `.m2` do *home* do usuário.

Para instalarmos o JAR do Cotuba no repositório local do Maven devemos, no diretório do projeto `cotuba-cli`, executar o seguinte comando:

```
# cotuba-cli

mvn clean install
```

O comando `mvn install` parte dos artefatos gerados pela fase `package` do Maven e, entre outras tarefas, os copia e cria as configurações necessárias no diretório apropriado do repositório local do Maven:

```
INFO] --- maven-install-plugin:2.4:install (default-install) @ cotuba-cli
---
[INFO] Installing .../cotuba/target/cotuba-cli-0.0.1-SNAPSHOT.jar to
.../.m2/repository/cotuba/cotuba-cli/0.0.1-SNAPSHOT/cotuba-cli-0.0.1-
SNAPSHOT.jar
[INFO] Installing .../cotuba/pom.xml to .../.m2/repository/cotuba/cotuba-
cli/0.0.1-SNAPSHOT/cotuba-cli-0.0.1-SNAPSHOT.pom
[INFO] Installing .../cotuba/target/cotuba-cli-0.0.1-SNAPSHOT-
distribution.zip to .../.m2/repository/cotuba/cotuba-cli/0.0.1-
SNAPSHOT/cotuba-cli-0.0.1-SNAPSHOT-distribution.zip
```

Com o JAR do Cotuba disponibilizado como dependência do Maven, o time da Paradizo pode gerar o JAR do projeto `tema-paradizo`

executando o comando:

```
# tema-paradizo

mvn clean package
```

Então, as pessoas da Paradizo podem copiar o JAR de tema para o diretório `libs`, que contém as dependências do Cotuba:

```
# tema-paradizo

cp target/tema-paradizo-*.jar ~/Desktop/libs/
```

Feito isso, a equipe da Paradizo pode ir ao `Desktop` e gerar um PDF com o comando:

```
./cotuba.sh -d solid-na-pratica/cotuba/livro-exemplo -f pdf
```

O PDF é gerado com sucesso! Porém, ao contrário do que esperávamos, o **tema não foi aplicado**. Por que será?

Para saber mais: repositórios remotos no Maven

Caso não estejamos trabalhando com um projeto open-source ou desejamos evitar que nossos usuários tenham de fazer o build do nosso projeto, podemos fornecer os artefatos das dependências em um repositório remoto do Maven.

O repositório padrão é o Maven Central Repository, que fica em https://repo1.maven.org/maven2/. Se um artefato não for encontrado nem no repositório local nem em nenhum outro repositório configurado, o Maven Central será o último a ser pesquisado. É o caso de dependências do Cotuba como Commonmark e Commons CLI, que são obtidas do repositório central do Maven.

Porém, podemos utilizar softwares como Sonatype Nexus ou JFrog Artifactory como repositórios de artefatos e configurá-los no `pom.xml` do nosso projeto. Por exemplo, a biblioteca que usamos parar gerar PDFs, a iText, disponibiliza seus artefatos por meio de um JFrog Artifactory: https://repo.itextsupport.com/. É possível que sejam

configuradas credenciais de autenticação, o que permite que tenhamos artefatos visíveis apenas em projetos da nossa organização.

O próprio GitHub pode ser utilizado como repositórios de artefatos. É o que o autor da biblioteca Epublib faz: https://github.com/psiegman/mvn-repo/raw/master/releases

6.3 Ligando os pontos (de extensão) com a Service Loader API

Temos um ponto de extensão no Cotuba através da interface `Plugin`. Temos uma implementação desse ponto de extensão através da classe `TemaParadizo`. Mas como ligar uma coisa com a outra, sem fazer com que o Cotuba dependa do código da Paradizo? A ideia é que as implementações dos pontos de extensão do Cotuba sejam aplicadas pela simples presença de seus JARs.

A Service Loader API

Nas versões iniciais da plataforma Java, para ligar um plugin de uma aplicação a uma implementação era necessário:

- criar uma solução caseira usando a Reflection API
- usar bibliotecas como JPF (http://jpf.sourceforge.net) ou PF4J (https://github.com/pf4j/pf4j)
- adotar uma especificação robusta, mas complexa, como OSGi (https://osgi.org/)

Porém, a partir do Java SE 6, a própria JRE contém uma solução: a **Service Loader API**. Na *Service Loader API*, um ponto de extensão é chamado de *service*. Para provermos um service precisamos de:

- **Service Provider Interface (SPI)**: interfaces ou classes abstratas que definem a assinatura do ponto de extensão.
- **Service Providers**: uma ou mais implementações da SPI.

No nosso caso, a interface `Plugin` é uma SPI que tem a classe `TemaParadizo` como um de seus service providers. Para ligar o service provider com sua SPI, o JAR do provider precisa definir o *provider configuration file*: um arquivo com o nome da SPI dentro da pasta `META-INF/services`. O conteúdo desse arquivo deve ser o *fully*

qualified name da implementação, contendo tanto o nome do pacote como o da classe. Como uma mesma SPI pode ter vários service providers, há bastante flexibilidade!

Com a Service Loader API, a simples presença de um `.jar` que implemente a abstração do plugin (ou SPI) fará com que o comportamento da aplicação seja estendido, sem precisarmos modificar nenhuma linha de código. É o **OCP ao extremo**!

Ligando o Service Provider com a SPI

A equipe da Paradizo deve fazer com que, no projeto `tema-paradizo`, o service provider `TemaParadizo` seja ligado à SPI `Plugin` do Cotuba. Para isso, deve ser criado, no diretório `src/main/resources`, o subdiretório `META-INF` e, dentro desse, o `services`.

Dentro do novo diretório `src/main/resources/META-INF/services`, deve ser criado um arquivo `cotuba.plugin.Plugin` (assim mesmo, com os pontos). O conteúdo desse arquivo deve ser o nome completo do service provider: `tema-paradizo/src/main/resources/META-INF/services/cotuba.plugin.Plugin`

```
br.com.paradizo.tema.TemaParadizo
```

Definido o arquivo de configuração do service provider, o time da Paradizo precisa regerar o JAR do `tema-paradizo` e copiá-los para o diretório `libs` do Cotuba com os comandos:

```
# tema-paradizo

mvn clean package
cp target/tema-paradizo-*.jar ~/Desktop/libs/
```

Porém, o tema ainda não é aplicado quando um PDF é gerado. O time do Cotuba precisa usar os service providers!

Carregando service providers no Cotuba

No projeto que define a SPI, devemos carregar os service providers usando a classe `java.util.ServiceLoader`. A classe `ServiceLoader` possui o método estático `load` que recebe uma SPI como parâmetro e, depois de vasculhar os diretórios `META-INF/services` dos JARs disponíveis no Classpath, retorna uma instância de `ServiceLoader`, que contém todas as implementações.

O `ServiceLoader` é um `Iterable` e, por isso, pode ser percorrido com um *for-each*. Caso não haja nenhum service provider para a SPI, o `ServiceLoader` se comporta como uma lista vazia.

Na interface `Plugin` do projeto `cotuba-cli`, vamos definir um método estático `renderizou` que usa a classe `ServiceLoader` para carregar os service providers disponíveis, invocando o método `aposRenderizacao` com um capítulo que acabou de ser renderizado:

```
// cotuba.plugin.Plugin

import cotuba.domain.Capitulo; // inserido
import java.util.ServiceLoader; // inserido

public interface Plugin {

  // código omitido...

  // inserido
  static void renderizou(Capitulo capitulo) {
    ServiceLoader.load(Plugin.class)
      .forEach(plugin -> {
        String html = capitulo.getConteudoHTML();
        String htmlModificado = plugin.aposRenderizacao(html);
        capitulo.setConteudoHTML(htmlModificado);
      });
  }

}
```

Precisamos invocar o novo método que aplica o hook de renderização assim que os arquivos Markdown de um capítulo forem transformados em HTML. Para isso, vamos alterar o seguinte trecho da classe `RenderizadorMDParaHTML`:

```
// cotuba.md.RenderizadorMDParaHTML

String html = renderer.render(document);
capitulo.setConteudoHTML(html);

Plugin.renderizou(capitulo); // inserido
```

Vamos criar também, na interface `Plugin`, um método estático semelhante ao criado anteriormente para ser invocado no fim da geração do ebook:

```
// cotuba.plugin.Plugin

public interface Plugin {

  // código omitido...

  // inserido
  static void gerou(Ebook ebook) {
    ServiceLoader.load(Plugin.class)
      .forEach(plugin -> plugin.aposGeracao(ebook));
  }

}
```

Assim que o ebook foi gerado, vamos invocar o novo método estático:

```
// cotuba.application.Cotuba

gerador.gera(ebook);

Plugin.gerou(ebook); // inserido
```

Pronto! Agora o projeto Cotuba invoca os plugins nos momentos apropriados!

DESAFIO

O método estático `load`, da classe `ServiceLoader`, vasculha o Classpath em busca de implementações da SPI, o que é um processo lento. Como otimizar a execução dos métodos `renderizou` e `gerou`?

Aplicando o plugin de tema

Digamos que a equipe da Paradizo obteve o código da versão atualizado do Cotuba, que executa corretamente os plugins. Será necessário fazer o build do Cotuba novamente:

```
# cotuba-cli

mvn clean install
```

Também será necessário descompactar o ZIP do Cotuba de novo:

```
# cotuba-cli

unzip -o target/cotuba-*-distribution.zip -d ~/Desktop
```

É preciso garantir que o JAR do projeto `tema-paradizo` esteja no diretório `libs` do Cotuba. Para isso, deve ser gerado o JAR do `tema-paradizo` com o comando:

```
# tema-paradizo

mvn clean package
```

O JAR gerado para o projeto `tema-paradizo` deve ser copiado para o diretório `libs`, que contém as dependências do Cotuba:

```
# tema-paradizo

cp target/tema-paradizo-*.jar ~/Desktop/libs/
```

Feito isso, o time da Paradizo pode tentar executar novamente o Cotuba, mas obterá o seguinte erro:

```
Exception in thread "main" java.lang.NoClassDefFoundError: org/jsoup/Jsoup
  at br.com.paradizo.tema.TemaParadizo.aplicaTema(TemaParadizo.java:23)
  at
br.com.paradizo.tema.TemaParadizo.aposRenderizacao(TemaParadizo.java:12)
  at cotuba.plugin.Plugin.lambda$renderizou$0(Plugin.java:18)
  ...
```

Além do JAR do `tema-paradizo`, é necessário o JAR da dependência utilizada para manipular o HTML: a biblioteca JSoup. Isso pode ser feito copiando o artefato do repositório local do Maven para os `libs` do Cotuba com o seguinte comando:

```
cp ~/.m2/repository/org/jsoup/jsoup/1.11.2/jsoup-1.11.2.jar
~/Desktop/libs/
```

Feito isso, a equipe da Paradizo pode gerar um PDF com o comando:

```
./cotuba.sh -d solid-na-pratica/cotuba/livro-exemplo -f pdf
```

Pronto! O tema foi aplicado! O mesmo deve acontecer na geração de outros formatos.

Para que serve OO?

Modelagem e Dependências

Você aprendeu Orientação a Objetos.

Entendeu classes, objetos, atributos, métodos, herança, polimorfismo, interfaces.

Aprendeu algumas soluções comuns para problemas recorrentes estudando alguns *Design Patterns*.

Figura 6.2: PDF com Tema Paradizo aplicado

FAT JARS

Para evitar a necessidade de copiar cada um dos JARs das bibliotecas utilizadas na hora de executar o Cotuba, o time da Paradizo poderia configurar o Maven para gerar um *fat JAR*: um JAR que inclui o projeto e todas as suas dependências. Para isso, poderiam ser utilizados o `maven-assembly-plugin` ou o `maven-shade-plugin` . Fica como um desafio para você!

6.4 Alguns usos da Service Loader API na plataforma Java

O mecanismo de Service Provider já existia internamente (http://lampwww.epfl.ch/java/jdk1.3/docs/guide/jar/jar.html#Service%20Provider) desde a JDK 1.3. A partir do Java SE 6, a Service Loader API ficou pública e disponível para aplicações. A API passou a ser usada em diferentes especificações da plataforma Java.

Drivers JDBC

Antes do Java SE 6, era necessário carregar programaticamente a implementação da interface `java.sql.Driver` de um driver JDBC. Para isso, antes de obter uma conexão, usávamos um código parecido com o seguinte:

```
Class.forName("com.mysql.jdbc.Driver");
```

Esse código aparentemente inútil tinha, como efeito colateral, o carregamento das implementações de `Driver` que seriam usadas posteriormente pela classe `DriverManager`.

Do Java SE 6 em diante, a classe `DriverManager` usa a Service Loader API para carregar automaticamente todos os drivers na inicialização. A chamada anterior ao método `forName` de `Class` passou a ser desnecessária (https://docs.oracle.com/javase/6/docs/api/java/sql/DriverManager.html). A interface `java.sql.Driver` passou a ser uma SPI.

No caso do MySQL, o arquivo `mysql-connector-java.jar` passou a ter o arquivo `META-INF/services/java.sql.Driver` com o seguinte conteúdo:

```
com.mysql.jdbc.Driver
```

Um fato interessante é que o Tomcat 7+ desliga o carregamento automático de drivers via Service Loader API para evitar vazamento de memória (http://tomcat.apache.org/tomcat-7.0-doc/jndi-datasource-examples-howto.html#DriverManager,_the_service_provider_mechanism_and_memory_leaks).

PersistenceProvider do JPA

É possível usar o JPA em uma aplicação Java SE, fora de um servidor de aplicação Java EE. Para isso, precisamos de uma implementação da interface `EntityManager`. Mas como instanciar essa abstração?

```
EntityManager manager = // ???
```

A especificação JPA determina que devemos usar a interface `EntityManagerFactory`. Porém, o problema continua: como instanciar a Factory?

```
EntityManagerFactory factory = // ???
EntityManager manager = factory.createEntityManager();
```

Pela especificação, para criar uma `EntityManagerFactory`, devemos usar um método estático da classe `Persistence`, passando o nome de uma *Persistence Unit*:

```
EntityManagerFactory factory =
    Persistence.createEntityManagerFactory("financas");
EntityManager manager = factory.createEntityManager();
```

Todas essas classes e interfaces são do pacote `javax.persistence`. Como ligá-las com uma implementação do JPA, como o Hibernate ou o Eclipse Link? Por meio da SPI `javax.persistence.spi.PersistenceProvider`.

O Hibernate tem o arquivo `META-INF/services/javax.persistence.spi.PersistenceProvider` dentro do `hibernate-core.jar`, com a implementação específica do Hibernate para a SPI:

```
org.hibernate.jpa.HibernatePersistenceProvider
```

Já o `eclipselink.jar` terá, no mesmo arquivo `META-INF/services/javax.persistence.spi.PersistenceProvider`, o service provider do Eclipse Link:

```
org.eclipse.persistence.jpa.PersistenceProvider
```

É interessante notar que a especificação JPA 1.0, parte da EJB 3.0 e Java EE 5, foi lançada para uso com o J2SE 5.0. Portanto, a Service Loader API ainda não era pública. O mecanismo de disponibilização das implementações era responsabilidade do JPA Provider.

Configuração programática da Servlet 3.0

A partir da especificação Servlet 3.0, parte do Java EE 6, é possível configurar Servlets e Filters sem ter que digitar várias linhas no `web.xml`. Podemos usar as anotações `@WebServlet` e `@WebFilter` em cima de nossas classes. Mas e para Servlets e Filters de frameworks e bibliotecas, cujo código não conseguimos modificar? Estamos fadados ao `web.xml`?

Há a SPI `javax.servlet.ServletContainerInitializer`, que define o método `onStartup`. Um Servlet Container compatível com a Servlet 3.0, como o Tomcat 7+ ou Jetty 8+, usa a Service Loader API para carregar e executar as implementações de `ServletContainerInitializer` na inicialização do servidor.

Por exemplo, o Spring tem o arquivo `META-INF/services/javax.servlet.ServletContainerInitializer` em seu `spring-web.jar`, com o seguinte conteúdo:

```
org.springframework.web.SpringServletContainerInitializer
```

Esse mecanismo faz com que o Spring dispare chamadas a classes sem a necessidade de configuração XML, como a classe `AbstractAnnotationConfigDispatcherServletInitializer .`

6.5 Um plugin para calcular estatísticas do ebook

Quais as palavras mais comuns no ebook? A empresa Cognitio quer saber! Para implementar a contagem de palavras, usarão o método `aposGeracao` de `Plugin`, a SPI do Cotuba, que é chamado após a geração do livro. O time da Cognitio deve criar um projeto Maven chamado `estatisticas-ebook` com as seguintes configurações:

- *Group Id*: `br.com.cognitio`
- *Artifact Id*: `estatisticas-ebook`
- a *Version* como `0.0.1-SNAPSHOT`
- o *Packaging* como `jar`
- a codificação de caracteres como UTF-8
- o Java 17 tanto para o código fonte como para o código compilado nos `.class`

A equipe de desenvolvimento da Cognitio também deve adicionar como dependências:

- o Cotuba, usada para implementar a SPI `Plugin`
- a biblioteca JSoup, usada para extrair as palavras dos HTMLs do capítulos

O `pom.xml` resultante deve ser parecido com o seguinte:

```
<!-- estatisticas-ebook/pom.xml -->

<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>br.com.cognitio</groupId>
  <artifactId>estatisticas-ebook</artifactId>
```

```
  <version>0.0.1-SNAPSHOT</version>
  <packaging>jar</packaging>

  <properties>
    <project.build.sourceEncoding>UTF-8
        </project.build.sourceEncoding>
    <maven.compiler.source>17</maven.compiler.source>
    <maven.compiler.target>17</maven.compiler.target>
  </properties>

  <dependencies>

    <dependency>
      <groupId>cotuba</groupId>
      <artifactId>cotuba-cli</artifactId>
      <version>0.0.1-SNAPSHOT</version>
    </dependency>

    <dependency>
      <groupId>org.jsoup</groupId>
      <artifactId>jsoup</artifactId>
      <version>1.11.2</version>
    </dependency>

  </dependencies>

</project>
```

Em seguida, o time da Cognitio deve criar, no pacote `br.com.cognitio.estatisticas`, a classe `CalculadoraDeEstatisticas`. Essa classe será um service provider para a SPI `Plugin` do Cotuba, definindo os métodos necessários. Não estamos interessados no método `aposRenderizacao`, mas é necessário retornar uma `String`. Por isso, vamos retornar o próprio HTML recebido:

```
// br.com.cognitio.estatisticas.CalculadoraDeEstatisticas

package br.com.cognitio.estatisticas;

import cotuba.domain.Ebook;
```

```
import cotuba.plugin.Plugin;

public class CalculadoraDeEstatisticas implements Plugin {

  @Override
  public String aposRenderizacao(String html) {
    return html;
  }

  @Override
  public void aposGeracao(Ebook ebook) {

  }

}
```

Para descobrir quais são as palavras mais recorrentes do ebook, será utilizado o método `aposGeracao`, no qual os desenvolvedores da Cognitio precisam:

- percorrer a lista de capítulos, obtendo o HTML de cada capítulo;
- remover as tags HTML, deixando só o texto, com a ajuda da biblioteca JSoup;
- o texto obtido deve ser quebrado usando espaços em branco como separador.

O método `split` de `String` quebra um texto de acordo com uma expressão regular. A expressão regular `\s+` pega vários espaços adjacentes. Por enquanto, o time da Cognitio apenas listará as palavras. A contagem em si ficará para depois.

```
// br.com.cognitio.estatisticas.CalculadoraDeEstatisticas

import cotuba.domain.Capitulo; // inserido

import org.jsoup.Jsoup; // inserido
import org.jsoup.nodes.Document; // inserido

public class CalculadoraDeEstatisticas implements Plugin {
```

```
  // código omitido...

  @Override
  public void aposGeracao(Ebook ebook) {

    for (Capitulo capitulo : ebook.getCapitulos()) {

      String html = capitulo.getConteudoHTML();

      Document doc = Jsoup.parse(html);

      String textoDoCapitulo = doc.body().text();

      String[] palavras = textoDoCapitulo.split("\\s+");

      for (String palavra : palavras) {
        System.out.println(palavra);
      }

    }

  }

}
```

Para que a classe `CalculadoraDeEstatisticas` realmente seja um service provider para a SPI `Plugin`, as pessoas de desenvolvimento da Cognitio devem criar, no diretório `src/main/resources` do projeto `estatisticas-ebook`, os subdiretórios `META-INF` e `services`. Dentro do novo diretório `services`, devem adicionar um arquivo `cotuba.plugin.Plugin`, definido o nome do service provider:

`estatisticas-ebook/src/main/resources/META-INF/services/cotuba.plugin.Plugin`

```
br.com.cognitio.estatisticas.CalculadoraDeEstatisticas
```

Mostrando palavras com o plugin de estatísticas

A equipe da Cognitio precisa obter o código e realizar o build do projeto `cotuba-cli`:

```
# cotuba-cli

mvn clean install
```

O ZIP do Cotuba deve ser descompactado em algum diretório, por exemplo, o Desktop:

```
# cotuba-cli

unzip -o target/cotuba-*-distribution.zip -d ~/Desktop
```

Também deve ser gerado o JAR do projeto `estatisticas-ebook` com o comando:

```
# estatisticas-ebook

mvn clean package
```

O JAR gerado para o plugin de estatísticas e a biblioteca JSoup devem ser copiados para o diretório `libs` do Cotuba:

```
# estatisticas-ebook

cp target/estatisticas-ebook-*.jar ~/Desktop/libs/
cp ~/.m2/repository/org/jsoup/jsoup/1.11.2/jsoup-1.11.2.jar
    ~/Desktop/libs/
```

Então, o PDF pode ser gerado:

```
./cotuba.sh -d solid-na-pratica/cotuba/livro-exemplo -f pdf
```

A presença do JAR do projeto `estatisticas-ebook` faz com que cada palavra do ebook seja impressa. As últimas linhas terão algo como:

```
MARTIN,
Robert
Cecil.
Getting
a
```

SOLID
start..
2009.
Em:
https://sites.google.com/site/unclebobconsultingllc/getting-a-solid-start.
Arquivo gerado com sucesso: book.pdf

Foram impressas as palavras do último capítulo do livro de exemplo, que contém referências bibliográficas. Os dados ainda não foram tratados e, por isso, ainda há pontuação. Além disso, ainda não foi feita a contagem das palavras.

6.6 O que aprendemos?

Neste capítulo, vimos como usar a Service Provider API do próprio Java para implementar dois plugins: um que aplica temas nos capítulos após a renderização dos HTMLs e outro que imprime todas as palavras assim que o ebook foi gerado. A simples presença do JAR desses plugins altera o comportamento do software. Não precisamos recompilar o Cotuba para que os temas sejam aplicados ou para que as palavras sejam impressas. O Cotuba está fechado para modificação, mas aberto para extensão! É o OCP potencializado!

No capítulo seguinte, vamos acompanhar a equipe de desenvolvimento da Cognitio na implementação do plugin de cálculo de estatísticas do ebook. Serão tomadas decisões que quebrarão o Princípio de Substituição de Liskov. Até lá!

O código deste capítulo pode ser encontrado em: https://github.com/alexandreaquiles/solid-na-pratica/tree/cap6-plugins

Figura 6.3: Vídeo do capítulo

CAPÍTULO 7

Princípio de Substituição de Liskov (LSP): herança do jeito certo

7.1 Continuando a implementação das estatísticas

A empresa Cognitio ainda não terminou a implementação do plugin de estatísticas do ebook. O projeto `estatisticas-ebook` está apenas listando as palavras encontradas. É necessário tratar os dados, removendo pontuação e acentuação, além de fazer com que não haja diferenciação entre letras maiúsculas e minúsculas. Depois disso tudo, vem o cálculo da nossa estatística inicial: a contagem das palavras.

Removendo pontuação

Ainda são impressas pontuações como `.`, `?`, `:`, `(`, `)`, entre outros. O método `replaceAll` de `String` recebe uma expressão regular. Nas expressões regulares do Java, é possível usar classes de caracteres compatíveis com o POSIX, um conjunto de especificações para sistemas operacionais. No POSIX, a classe de caracteres `\p{Punct}` é equivalente a qualquer um dos caracteres a seguir:

```
!"#$%&'()*+,-./:;<=>?@[\]^_`{|}~
```

Com isso em mente, a equipe da Cognitio pode trocar pontuações por espaços com o seguinte código:

```
String textoDoCapituloSemPontuacao =
    textoDoCapitulo.replaceAll("\\p{Punct}", " ");
```

Removendo acentuação

Outra coisa que importante é limpar acentuação, trocando os caracteres acentuados pelas versões não acentuadas. Por exemplo, trocaríamos `ê` por `e`, `ú` por `u` e `ç` por `c`.

Para isso, é possível usar a classe `java.text.Normalizer`, disponível a partir do Java 6. O método `normalize` dessa classe, pode ser usado com diferentes formas de normalização. A forma de normalização D, ou NFD, faz uma decomposição canônica: retorna uma `String` cujos caracteres são decompostos de acordo com o padrão Unicode.

Por exemplo, uma `ç` no Unicode pode ser representada pelo caractere `\u00e7`, o *LATIN SMALL LETTER C WITH CEDILLA*. Mas o mesmo caractere normalizado com NFD é decomposto em um `\u0063`, um *LATIN SMALL LETTER C* (um `c`), seguido por um `\u0327`, um *COMBINING CEDILLA*.

Com os caracteres decompostos, é possível remover todos os que não forem letras sem acentuação nem pontuação básica. Para isso, pode ser usada a classe de caracteres POSIX `\p{ASCII}`.

A partir da `String` decomposta, o time da Cognitio pode remover tudo o que *não* estiver na classe `\p{ASCII}` com a expressão regular `[^\p{ASCII}]`:

```
String decomposta =
    Normalizer.normalize(textoDoCapituloSemPontuacao,
        Normalizer.Form.NFD).
String textoDoCapituloSemAcentos =
    decomposta.replaceAll("[^\\p{ASCII}]", "");
```

Tudo maiúsculo

Para a contagem, palavras com letras maiúsculas não são diferentes de minúsculas. As pessoas do desenvolvimento da Cognitio podem ignorar essa diferença passando todas as letras para maiúsculas:

```
String emMaiusculas = palavra.toUpperCase();
```

Uma contagem preliminar

Para contar as palavras, uma ideia é usar um `Map`, associando uma palavra, uma `String`, ao seu número de ocorrências, um `Integer`:

```
Map<String, Integer> contagemDePalavras;
```

É possível usar um `HashMap` como implementação. Mas, como um `HashMap` não tem garantias de qual chave vem primeiro, as palavras seriam exibidas em uma ordem estranha. Para que as chaves sejam mantidas ordenadas, é interessante usar um `TreeMap`:

```
Map<String, Integer> contagemDePalavras = new TreeMap<>();
```

O `TreeMap` usa internamente uma árvore de busca binária balanceada que mantém as chaves ordenadas de maneira eficiente. Busca, inserção e remoção são feitas em *O(log n)*. A cada inserção, as chaves são mantidas em *ordem natural*: ordem crescente para números, ordem alfabética para textos. Para objetos, é necessário implementar a interface `Comparable`, que define o método `compareTo`.

Como as chaves são as palavras, que são do tipo `String`, será possível percorrê-las em ordem alfabética. Mas e a contagem em si? Um esboço inicial é iniciar toda palavra encontrada com uma ocorrência:

```
contagemDePalavras.put(palavra, 1);
```

> **O PODER DE BOAS ABSTRAÇÕES**
>
> Vale notar que a API de Collections do próprio Java mostra o poder de uma boa abstração.
>
> A interface `Map` abstrai uma estrutura de dados que mantém uma associação chave-valor. Essa abstração é implementada por diferentes classes: `HashMap`, `LinkedHashMap` e `TreeMap`.
>
> Há garantias de um comportamento comum, mas cada implementação tem comportamentos próprios que trazem bastante flexibilidade para quem as usa.

Percorrendo as entradas do Map

É possível percorrer os valores de dentro de um `Map` usando o *entry set*, que disponibiliza a cada iteração um `Map.Entry` que representa cada par chave-valor (*key-value*). No caso do `Map` de contagem de palavras, a *key* seria a palavra e o *value* seria a contagem dessa palavra.

A equipe da Cognitio poderia percorrer as palavras e mostrá-la associada ao seu número de ocorrências com o seguinte trecho de código:

```
for (Map.Entry<String, Integer> contagem :
    contagemDePalavras.entrySet()) {

  String palavra = contagem.getKey();

  Integer ocorrencias = contagem.getValue();

  System.out.println(palavra + ": " + ocorrencias);
}
```

Avançando na implementação da contagem de palavras

Juntando tudo, a implementação do método `aposGeracao` da classe `CalculadoraDeEstatisticas` ficaria algo como:

```
// br.com.cognitio.estatisticas.CalculadoraDeEstatisticas

// código omitido...

String textoDoCapitulo = doc.body().text();

// inserido
String textoDoCapituloSemPontuacao =
  textoDoCapitulo.replaceAll("\\p{Punct}", " ");

// inserido
String textoDoCapituloSemAcentos =
  Normalizer.normalize(textoDoCapituloSemPontuacao,
      Normalizer.Form.NFD)
  .replaceAll("[^\\p{ASCII}]", "");
```

```
String[] palavras = textoDoCapituloSemAcentos.split("\\s+");

for (String palavra : palavras) {

  String emMaiusculas = palavra.toUpperCase(); // inserido

  System.out.println(palavra);

}
```

No topo do método `aposGeracao` de `CalculadoraDeEstatisticas`, é necessário que seja instanciado um `TreeMap`. Cada palavra encontrada deve ser inserida como chave do `Map`, associando ao valor `1`, por enquanto.

```
// br.com.cognitio.estatisticas.CalculadoraDeEstatisticas

@Override
public void aposGeracao(Ebook ebook) {

  // inserido
  Map<String, Integer> contagemDePalavras = new TreeMap<>();

  for (Capitulo capitulo : ebook.getCapitulos()) {

    // código omitido...

    String[] palavras = textoDoCapituloSemAcentos.split("\\s+");

    for (String palavra : palavras) {

      String emMaiusculas = palavra.toUpperCase();

      contagemDePalavras.put(emMaiusculas, 1); // inserido

    }

  }
```

```
}
```

Ao final do método `aposGeracao`, depois do `for` que percorre os capítulos, o `entry set` do `Map` deve ser percorrido, imprimindo cada palavra encontrada e o respectivo número de ocorrências:

```
// br.com.cognitio.estatisticas.CalculadoraDeEstatisticas

for (Map.Entry<String, Integer> contagem :
    contagemDePalavras.entrySet()) {

  String palavra = contagem.getKey();

  Integer ocorrencias = contagem.getValue();

  System.out.println(palavra + ": " + ocorrencias);
}
```

7.2 Contando palavras

Se a equipe da Cognitio executar o plugin de estatísticas nesse momento, todas as palavras do ebook estão sendo impressas em letras maiúsculas, sem pontuação nem acentuação. Porém, a contagem ainda não está sendo feita. É como se cada palavra fosse única:

```
WITH: 1
WITHIN: 1
WITHOUT: 1
WOULD: 1
WRITE: 1
WWW: 1
```

O problema é que ainda não foi implementada a contagem de palavras. Um `Map`, mais especificamente um `TreeMap`, **não faz contagem nem acumula valores**. Cada chave de um `Map` só pode ter um valor associado. Se a palavra já existe, é associado o valor `1` novamente.

Implementando a contagem de palavras com herança

Depois de muita deliberação, o time da Cognitio decidiu criar uma versão de `TreeMap` que mantém a contagem de valores repetidos associados a cada chave. A solução implementada define uma classe `ContagemDePalavras` que usa **herança** para criar um tipo de `TreeMap`:

```
// br.com.cognitio.estatisticas.ContagemDePalavras

package br.com.cognitio.estatisticas;

import java.util.TreeMap;
```

```
class ContagemDePalavras extends TreeMap<String, Integer> {

}
```

A classe `ContagemDePalavras` contém um método `adicionaPalavra`, que incrementa a contagem caso a palavra esteja repetida. Na primeira ocorrência da palavra, é usado `1` como valor:

```
// br.com.cognitio.estatisticas.ContagemDePalavras

class ContagemDePalavras extends TreeMap<String, Integer> {

  void adicionaPalavra(String palavra) {

    Integer contagem = get(palavra);

    if (contagem != null) {
      contagem++;
    } else {
      contagem = 1;
    }

    put(palavra, contagem);

  }

}
```

Em `CalculadoraDeEstatisticas`, deve ser usada a nova classe:

```
// br.com.cognitio.estatisticas.CalculadoraDeEstatisticas

Map<String, Integer> contagemPalavras = new TreeMap<>();
var contagemDePalavras = new ContagemDePalavras();

for (Capitulo capitulo : ebook.getCapitulos()) {

  // código omitido...

  String[] palavras = textoDoCapituloSemAcentos.split("\\s+");
```

```
  for (String palavra : palavras) {

    String emMaiusculas = palavra.toUpperCase();

    contagemPalavras.put(emMaiusculas, 1);
    contagemDePalavras.adicionaPalavra(emMaiusculas);
  }

}
```

Os imports desnecessários devem ser removidos. No trecho da classe `CalculadoraDeEstatisticas` que imprime os resultados, o método `entrySet` deve continuar sendo compilado com sucesso na variável `contagemDePalavras` , já que a classe `ContagemDePalavras` é um `Map` e, por isso, possui esse método:

```
for (Map.Entry<String, Integer> contagem :
    contagemDePalavras.entrySet()) { // funciona!

  // código omitido...

}
```

Depois de feito o build do projeto `estatisticas-ebook` , copiado o JAR para o diretório `libs` do Cotuba e regerado o livro, as palavras são impressas com a contagem correta:

```
WITH: 1
WITHIN: 2
WITHOUT: 1
WOULD: 1
WRITE: 1
WWW: 2
```

Perceba que tanto `WITHIN` como `WWW` ocorrem 2 vezes no livro.

7.3 Herdando inutilidades

O cálculo de estatísticas funcionou. Mas o design do código não está satisfatório. Há um ponto bem ruim: será que a classe `ContagemDePalavras` deve mesmo herdar de `TreeMap` ?

O `extends` pode ser lido como *é um* ou *é um tipo especial de*. Não faz sentido falar que a classe `ContagemDePalavras` é um tipo especial de `TreeMap` . A classe `ContagemDePalavras` deve ser usada para adicionar palavras, acumulando a sua contagem. Não deve ser feito tudo o que é possível fazer com um `TreeMap` . Por exemplo, não são necessários os métodos:

- `clear` , que limpa o conteúdo;
- `remove` , que remove uma chave e seu valor associado;
- `replace` , que troca o valor associado a uma chave retornado o valor antigo (ou `null`).

Na verdade, a maioria dos métodos de `TreeMap` não tem utilidade para a classe `ContagemDePalavras` . Apenas alguns são realmente utilizados:

- os métodos `get` e `put` de `TreeMap` são usados internamente na classe `ContagemDePalavras` .
- o método `entrySet` é usado em um *for-each* na classe `CalculadoraDeEstatisticas` .

Porém, as classes que usam `ContagemDePalavras` , como `CalculadoraDeEstatisticas` , podem invocar qualquer método disponível em um `TreeMap` . Para evitar que esses métodos indesejados, é possível sobrescrevê-los e, se forem chamados, lançar uma exceção que sinalizada que a operação não é suportada:

```
// br.com.cognitio.estatisticas.ContagemDePalavras
```

```
class ContagemDePalavras extends TreeMap<String, Integer> {

  // código omitido...

  @Override
  public void clear() {
    throw new UnsupportedOperationException();
  }

  @Override
  public Integer remove(String key) {
    throw new UnsupportedOperationException();
  }

  @Override
  public Integer replace(String key, Integer value) {
    throw new UnsupportedOperationException();
  }

  // o mesmo para todos os outros métodos que não queremos usar...

}
```

Parece uma boa solução? Na verdade, parece que é usada uma parcela muito pequena da classe herdada.

OO não é só herança

Logo depois de uma introdução a OO, a ideia de herança é a que fica. Ao falar em OO, o que vem à mente, em geral, é: especialização, subclasses e sobrescrita de métodos.

A tentação de herdar de uma superclasse para reaproveitar código é muito forte. Mas um especialista em OO raramente usa esses recursos. Então, quando usar herança é justificado?

"Se um gato possui raça e patas, e um cachorro possui raça, patas e tipoDoPelo, logo `Cachorro extends Gato` *? Pode parecer engraçado, mas é (...) herança por preguiça, por comodismo, porque vai dar uma ajudinha. A relação "é um" não se encaixa aqui, e vai nos gerar problemas."*

Paulo Silveira, no artigo *Como não aprender orientação a objetos: Herança* (SILVEIRA, 2006)

7.4 O Princípio da Substituição de Liskov (LSP)

No artigo *Data Abstraction and Hierarchy* (LISKOV, 1988), a cientista da computação Barbara Liskov faz a seguinte afirmação sobre hierarquia de tipos: *"Se para cada objeto o1 do tipo S há um objeto o2 do tipo T e, para todos os programas P definidos em termos de T, o comportamento de P não é modificado quando o1 é substituído por o2, então S é um subtipo de T."*

Uma afirmação que usa uma linguagem matemática um tanto complexa de ler, não? No mesmo artigo, a autora descreve a mesma ideia de uma forma mais simples: *"A ideia intuitiva de um subtipo é aquela cujos objetos fornecem todo o comportamento de objetos de outro tipo (o supertipo) mais algo extra."*

Barbara Liskov dá exemplos de objetos que **não são** subtipos um do outro:

- Um conjunto não é um subtipo de uma lista, nem o contrário. Em um conjunto (como uma implementação de `Set` do Java) um mesmo elemento só pode ser adicionado uma vez. Já em uma lista (como um `List` do Java) é possível repetir elementos. Portanto, um programa que espera uma lista, mas recebe um conjunto pode apresentar um comportamento indesejado.
- Uma pilha não é um subtipo de uma fila. Quando removemos um elemento de uma pilha (como uma `Stack` do Java) o último item é removido, já que uma pilha é uma estrutura de dados LIFO (*Last In First Out*). Já uma fila (como uma `Queue` do Java) é FIFO (*First In First Out*), o que faz com que o primeiro elemento seja removido. O comportamento de um programa que usa uma pilha mudaria se fosse recebida uma fila.

Um detalhe importante é destacado pela autora: não basta que a assinatura dos métodos seja a mesma, as operações de um subtipo

devem ter o mesmo comportamento do supertipo.

Em seguida, Liskov mostra exemplos de objetos que **são** de fato subtipos:

- Arrays, sequências (como um `LinkedList` do Java) e conjuntos indexados (como um `LinkedHashSet` do Java) são subtipos de uma coleção indexada, já que possuem uma operação de busca por um elemento de uma determinada posição. Todos esses subtipos podem ter operações extras.
- Diferentes dispositivos de I/O são subtipos de um I/O abstrato que, por exemplo, tem um teste de *end-of-file* (fim de arquivo). Operações extras podem ser definidas em dispositivos específicos.

Uncle Bob resgata o trabalho de Barbara Liskov para cunhar um dos princípios SOLID:

> **LISKOV SUBSTITUTION PRINCIPLE (LSP)**
>
> *Subtipos devem ser substituíveis por seus tipos base.*

É o princípio da *substituibilidade*. Quando usamos herança do jeito certo, deve ser possível usar qualquer método das classes filhas ao recebermos uma variável cujo tipo é o da classe mãe.

Uma classe filha herda todos os comportamentos (métodos) da classe mãe. Porém, não podemos usar só parte desses comportamentos. Alguns comportamentos podem ser sobrescritos e modificados, mas não podem ser desligados nem removidos.

Podemos fazer mais coisas que a classe mãe e até alterar alguns comportamentos, mas não podemos fazer nada a menos.

7.5 Favorecendo composição à herança

Para atender ao LSP, `ContagemDePalavras` não deve herdar de `TreeMap` porque não há suporte a boa parte dos métodos da classe mãe. A classe `ContagemDePalavras` não é substituível por `TreeMap` pois faz menos coisas que sua superclasse.

Qual uma implementação melhor? Deve ser simplesmente utilizado um `TreeMap` como um atributo, evitando a herança:

```
// br.com.cognitio.estatisticas.ContagemDePalavras

import java.util.Map; // inserido

class ContagemDePalavras extends TreeMap<String, Integer> {

  private Map<String, Integer> map = new TreeMap<>(); // inserido

  // ...
}
```

Ter uma referência de um objeto como atributo privado, conforme o código anterior, é algo que chamamos de **composição**. Fazer uma composição com uma dependência é preferível a usar herança, criando um tipo especial dessa dependência.

Alguns livros clássicos de design OO contêm lemas alinhados com essa ideia:

- *Design Patterns* (GAMMA et al., 1994): *"Favoreça composição à herança."*
- *Effective Java* (BLOCH, 2001): *"Item 14: Prefira composição à herança."*

Agora que a classe `ContagemDePalavras` faz uma composição com o `TreeMap`, os métodos `adicionaPalavra` e `entrySet` devem ser

corrigidos:

```
// br.com.cognitio.estatisticas.ContagemDePalavras

class ContagemDePalavras {

  // código omitido...

  void adicionaPalavra(String palavra) {

    Integer contagem = get(palavra);
    Integer contagem = map.get(palavra);

    if (contagem != null) {
      contagem++;
    } else {
      contagem = 1;
    }

    put(palavra, contagem);
    map.put(palavra, contagem);

  }

}
```

No momento em que `ContagemDePalavras` deixa de ser um `TreeMap`, deve acontecer um erro de compilação na classe `CalculadoraDeEstatisticas`, que espera que haja um método `entrySet`. Uma solução simples, que adotaremos por enquanto, é criar esse método em `ContagemDePalavras`, retornando um `Set` com os pares chave-valor do `Map`:

```
// br.com.cognitio.estatisticas.ContagemDePalavras

import java.util.Set; // inserido

class ContagemDePalavras {

  // código omitido
```

```
  // inserido
  Set<Map.Entry<String, Integer>> entrySet() {
    return map.entrySet();
  }

}
```

Depois de feito o build do projeto `estatisticas-ebook`, copiado o JAR para as `libs` do Cotuba e gerado um ebook, as palavras devem continuar sendo exibidas corretamente!

> Uma solução alternativa seria utilizar uma estrutura de dados já pronta de alguma biblioteca que cuidaria exatamente do problema de contar quantas vezes um mesmo elemento é adicionado em um conjunto.
>
> É o que um *Multiset* da biblioteca Google Guava se propõe a fazer: https://guava.dev/releases/snapshot/api/docs/com/google/common/collect/Multiset.html

A forte intimidade entre uma (classe) filha e sua mãe

No livro *Refactoring* (FOWLER et al., 1999), Kent Beck e Martin Fowler listam alguns problemas de design comuns, que chamam de maus cheiros de código. Entre esses maus cheiros, há a *intimidade inadequada*: *"Às vezes as classes se tornam íntimas demais e gastam tempo demais sondando as partes privadas das outras. Podemos não ser pudicos quando o assunto são pessoas, mas achamos que nossas classes devem seguir regras puritanas rígidas. A herança pode muitas vezes levar à intimidade excessiva. Subclasses sempre saberão mais sobre seus pais do que esses gostariam que elas soubessem."*

O uso de herança faz com que a classe filha tenha um forte acoplamento (ou intimidade) com sua classe mãe.

Esse forte acoplamento faz com que especialistas em OO como Joshua Bloch, no livro *Effective Java* (BLOCH, 2001), indique que a maioria das classes de uma aplicação deveria proibir herança: *"Item 15: Faça um design e documente pensando em herança ou proíba-a."* No Java, com o uso de `final` antes do nome da classe, é possível fazer com que nenhuma outra classe consiga herdá-la.

Joshua Bloch sugere, no mesmo livro, que evitemos herança para abstrações: *"Item 16: Prefira interfaces a classes abstratas."*

Classes abstratas podem definir atributos, métodos concretos e métodos abstratos, que contêm apenas assinaturas. Interfaces são mais leves: apenas definem assinaturas de métodos que devem ser implementados. Por isso, há menos acoplamento.

Quando usar herança?

Joshua Bloch diz, ainda no mesmo livro, que herança só é apropriada em circunstâncias em que a subclasse é realmente um subtipo da superclasse, ou seja, se um relacionamento "é-um" existir entre as duas classes. Segundo o autor, o uso de herança deve ser precedido da pergunta: cada B é realmente um A? Quando a resposta é "não", geralmente é o caso em que B deve conter uma instância privada de A e expor uma API menor e mais simples: A não é uma parte essencial de B, apenas um detalhe de sua implementação.

No caso do Cotuba, poderíamos ter um tipo especial de ebook que, além de fazer tudo o que um ebook faz e conter tudo o que um ebook contém, ainda tem um método que adiciona propagandas em algumas páginas. Definiríamos uma classe `EbookComPropagandas` que herda de `Ebook`, por exemplo.

O uso incorreto de herança na plataforma Java

Paulo Silveira, no artigo *Como não aprender orientação a objetos: Herança* (SILVEIRA, 2006), cita alguns casos da própria plataforma

Java em que há mau uso de herança. Um dos exemplos de Paulo é a classe `java.util.Properties`:

```
public class Properties extends Hashtable<Object,Object> {
  //...
}
```

A classe `Properties`, definida no Java 1.0, herda de `Hashtable`. Isso traz efeitos indesejados, fazendo com que a chamada de alguns métodos não seja recomendada. A classe filha deseja fazer menos que sua mãe. Ou seja, há uma quebra do LSP.

Isso é explicado no próprio *Javadoc* (https://docs.oracle.com/javase/10/docs/api/java/util/Properties.html): *"Como `Properties` herda de `Hashtable`, os métodos `put` e `putAll` podem ser aplicados a um objeto `Properties`. Seu uso é fortemente desencorajado, pois permite que sejam inseridas entradas cujas chaves ou valores não são `String`."*

Paulo dá outro exemplo usando a API de Servlets do Java EE. Para definir uma Servlet em um projeto Java para Web, devemos herdar de `javax.servlet.http.HttpServlet` e sobrescrever os métodos apropriados como `doGet` para requisições HTTP do tipo GET, `doPost` para POST ou `service` para qualquer tipo de requisição.

Para obter configurações na inicialização de uma Servlet, podemos sobrescrever o método `init`, que recebe um `ServletConfig`. Mas há inconsistências terríveis: no caso do `doGet`, `doPost` ou `service`, não podemos chamar o método da classe mãe, ocorre uma exceção. Já no caso do `init`, devemos obrigatoriamente chamar o método `init` da classe mãe, passando o `ServletConfig` como parâmetro.

```
public class OiServlet extends HttpServlet {

  @Override
  public void init(ServletConfig config)
      throws ServletException {
    super.init(config); //se NÃO chamar, dá erro...
  }
```

```
  @Override
  protected void service(HttpServletRequest req,
      HttpServletResponse res)
      throws IOException, ServletException {
    super.service(req, res); //se chamar, dá erro...
  }

}
```

7.6 Qual a necessidade de um método se não há nada para ser feito?

O Cotuba agora tem dois pontos de extensão:

- o hook de renderização do HTML, usado pela empresa Paradizo
- o hook de geração do ebook, usado pela empresa Cognitio para calcular estatísticas do ebook

Tudo compila e funciona perfeitamente! Mas há algo de incômodo nos service providers, que implementam a SPI `Plugin`. Observe que nada é feito no método `aposGeracao` da classe `TemaParadizo`:

```
// br.com.paradizo.tema.TemaParadizo

public class TemaParadizo implements Plugin {

  // código omitido...

  @Override
  public void aposGeracao(Ebook ebook) {
    // NADA AQUI!
  }

}
```

Já na classe `CalculadoraDeEstatisticas`, é no método `aposRenderizacao` em que não nada é feito. Porém, como é necessário retornar uma `String`, o próprio HTML recebido é retornado. Dessa maneira, evitamos bugs!

```
// br.com.cognitio.estatisticas.CalculadoraDeEstatisticas

public class CalculadoraDeEstatisticas implements Plugin {
```

```
  @Override
  public String aposRenderizacao(String html) {
    return html;
  }

   // código omitido...

}
```

Será que dever ser lançadas exceções, como foi cogitado na classe de contagem de palavras? Para a classe `CalculadoraDeEstatisticas`, seria algo como:

```
// br.com.cognitio.estatisticas.CalculadoraDeEstatisticas

public class CalculadoraDeEstatisticas implements Plugin {

  @Override
  public String aposRenderizacao(String html) {

    throw new UnsupportedOperationException(
      "Não há suporte ao plugin de renderização."); // uma ideia

  }

  // código omitido...

}
```

Uma `UnsupportedOperationException` também poderia ser lançada na classe `TemaParadizo`, no método `aposGeracao`. Se essa solução de lançar a exceção for adotada, a geração de ebooks deixará de funcionar, já que todos os métodos dos service providers chegam a ser executados. Parece que continuar não fazendo nada nesses métodos é uma solução melhor!

Mas qual é a causa raiz de termos que definir métodos que não usamos? Por qual motivo, dependendo de quem estiver implementando a interface `Plugin`, só faz sentido definir uma coisa ou outra?

No fim das contas, há uma **quebra do LSP**: as implementações não são substituíveis pela interface, já que fornecem só parte do comportamento definido pela abstração. Como resolver? Isso é assunto para o próximo capítulo!

Liskov e exceções

É importante dizer que a própria Barbara Liskov considera que exceções são válidas de serem lançadas em subtipos em seu artigo *Data Abstraction and Hierarchy* (LISKOV, 1988).

Um dos exemplos de hierarquia de subtipos abordados por Liskov trata de dispositivos de I/O, que teriam operações abstratas de entrada e saída e seriam implementadas por dispositivos específicos.

Liskov menciona que uma impressora teria operações de modificação como `put_char`, mas não teria suporte a operações de leitura como `get_char`. Uma alternativa, descrita pela autora, é manter as operações `put_char` e `get_char` na abstração, indicando que uma exceção pode ser sinalizada. Quando invocado em uma impressora, a operação `get_char` sinalizaria uma exceção.

Neste livro, o LSP é tratado como uma versão mais estrita que o próprio artigo original de Liskov.

Para saber mais: violação do LSP nas Collections do Java

Na API de Collections do Java, é possível criar uma versão não modificável de uma lista a partir do método `Collections.unmodifiableList`. A lista não modificável implementa todos os métodos mutáveis da interface `List`, como `add` e `set`, entre diversos outros. Porém, se um código que tem uma `List` como parâmetro receber uma implementação não modificável e

invocar algum dos métodos mutáveis, terá como resultado uma `UnsupportedOperationException`.

Além de listas, qualquer uma das *unmodifiable collections* do Java mantém os métodos mutáveis. Ou seja, uma coleção não modificável é declarada como um tipo especial da coleção mutável, mas algumas operações não são suportadas. Por isso, as coleções não modificáveis **não** são substituíveis pelos tipos que implementam e, portanto, quebram o LSP.

> Em um capítulo posterior, abordaremos imutabilidade e usaremos coleções não modificáveis do Java.

Já na linguagem Objective-C, o padrão são coleções imutáveis, que são herdadas por suas versões mutáveis. Por exemplo, o conjunto `NSSet` é imutável (https://developer.apple.com/documentation/foundation/nsset/) e suas funcionalidades são herdadas e estendidas por `NSMutableSet`, que adiciona métodos mutáveis (https://developer.apple.com/documentation/foundation/nsmutableset). Dessa maneira, o LSP é respeitado e não há a necessidade de exceções em operações indesejadas.

7.7 Contraponto: críticas ao LSP

Dan North, no artigo *CUPID - the back story* (NORTH, 2021), argumenta o LSP faz sentido se pensarmos em termos de subtipos que agem da mesma maneira que o código substituído. Mas é comum pensar o LSP em termos de subclasses que, em vez de ter uma relação "é um" com a superclasse, tem uma relação "age como um", "às vezes é usado como um" ou até mesmo "passa como um se você apertar os olhos". Nesse sentido, é preferível compor tipos simples em estruturas complexas.

David Copeland, no livro *SOLID is not solid* (COPELAND, 2019), critica o artigo original de Uncle Bob, que trata de detalhes como ponteiros e usa exemplos de quadrados e retângulos, mas não indica com clareza qual é o problema que está sendo resolvido nem qual solução deve ser adotada.

Kevlin Henney, na palestra *The SOLID Design Principles Deconstructed* (HENNEY, 2013), resgata o artigo *Data Abstraction and Hierarchy* (LISKOV, 1988) de Barbara Liskov, mostrando que a relação entre um subtipo e um supertipo é bem mais estrita que a interpretação comum do LSP. Segundo o palestrante, o artigo de Liskov trata de relações entre *Abstract Data Types* (ADTs) e não da relação de herança em linguagens OO. Henney indica que há duas versões do LSP: a estrita do artigo de Liskov e a ideal geral de "não quebrar o contrato" de uma superclasse.

7.8 O que aprendemos?

Neste capítulo, acompanhamos a equipe da Cognitio na implementação do cálculo de estatísticas do ebook. A solução adotada usava herança de uma maneira inadequada, fazendo com que nem todos os comportamentos da classe mãe fossem úteis para a classe filha. Aprendemos que isso é um sinal da quebra do LSP e vimos como trocar herança por composição resolve o problema.

Vimos ainda que a interface `Plugin` parece quebrar o LSP, já que nem todos os *service providers* seguem o contrato definido pela SPI. No próximo capítulo, veremos que a origem desse problema é uma quebra do Princípio da Segregação de Interfaces!

O código deste capítulo pode ser encontrado em: https://github.com/alexandreaquiles/solid-na-pratica/tree/cap7-liskov-substitution-principle

Figura 7.1: Vídeo do capítulo

Capítulo 8

Princípio da Segregação de Interfaces (ISP): clientes separados, interfaces separadas

A SPI `Plugin` do Cotuba é implementada tanto pelo *service provider* de temas da Paradizo como pelo *service provider* de estatísticas da Cognitio.

Porém, nem todos os métodos definidos na SPI são implementados de fato pelos *service providers*. No plugin de temas, não fazemos nada no método `aposGeracao`. No plugin de estatísticas, não fazemos nada no método `aposRenderizacao`, retornando o próprio HTML recebido.

Uma implementação não deveria fornecer apenas parte de uma abstração. É uma quebra do LSP. Mas a questão principal é que temos dois usos diferentes, temas e finalização da geração, para a mesma interface `Plugin`. Cada interface deveria ter um uso específico.

8.1 O Princípio da Segregação de Interfaces (ISP)

Uncle Bob trata do problema de diferentes usos para uma mesma interface definindo o seguinte princípio:

> **Interface Segregation Principle (ISP)**
>
> *Clientes não devem ser obrigados a depender de métodos que eles não usam.*

Não deveria haver a necessidade de retornar um valor nulo ou lançar uma exceção quando o contrato de uma interface não é atendido.

O artigo *Design Principles and Design Patterns* (MARTIN, 2000), também de Uncle Bob, diz: *"Muitas interfaces específicas para cada cliente são melhores que uma interface de propósito geral."*

De certa forma, o ISP trata de **coesão para interfaces**. No livro *OO e SOLID para Ninjas* (ANICHE, 2015), Maurício Aniche declara que interfaces coesas são aquelas cujos comportamentos são simples e bem definidos e que classes que dependem de interfaces leves sofrem menos com mudanças em outros pontos do sistema.

No fim das contas, trata-se de avaliar a qualidade de nossas abstrações. Devemos pensar se todo o contrato definido pela abstração faz sentido de ser usado como uma coisa só. Se o uso de uma abstração é diferente em diferentes classes, devemos ter mais de uma interface (ou classe abstrata).

> *"Clientes separados, interfaces separadas."*
>
> Uncle Bob, no livro *Agile Principles, Patterns, and Practices in C#* (MARTIN, 2006)

Para atender ao ISP no Cotuba, é necessário quebrar a SPI `Plugin` em duas:

- uma SPI específica para o hook de renderização do HTML
- uma outra SPI para o hook de geração de ebook

Uma interface separada para o hook de renderização

No pacote `cotuba.plugin` do projeto `cotuba-cli`, deve ser adicionada uma interface chamada `AoRenderizarHTML`. A nova interface deve definir um método `aposRenderizacao` semelhante ao da interface `Plugin`:

```
// cotuba.plugin.AoRenderizarHTML

package cotuba.plugin;

public interface AoRenderizarHTML {

  String aposRenderizacao(String html);

}
```

O método estático `renderizou` da interface `Plugin` deve ser movido para a interface `AoRenderizarHTML`. O tipo passado ao método `load` do `ServiceLoader` deve ser modificado para `AoRenderizarHTML`:

```
// cotuba.plugin.AoRenderizarHTML

import cotuba.domain.Capitulo; // inserido
import java.util.ServiceLoader; // inserido

public interface AoRenderizarHTML {
```

```
  String aposRenderizacao(String html);

  // inserido
  static void renderizou(Capitulo capitulo) {

    ServiceLoader.load(AoRenderizarHTML.class) // modificado!
      .forEach(plugin -> {
        // restante do código...
      });
  }

}
```

Logo depois da renderização do markdown de cada capítulo em HTML, na classe `RenderizadorMDParaHTML`, o método estático `renderizou` deve ser invocado a partir da interface `AoRenderizarHTML`:

```
// cotuba.md.RenderizadorMDParaHTML

String html = renderer.render(document);
capitulo.setConteudoHTML(html);

~~Plugin.renderizou(capitulo);~~
AoRenderizarHTML.renderizou(capitulo);
```

No projeto `tema-paradizo`, a equipe da Paradizo deve obter a nova versão do projeto `cotuba-cli` e fazer com que a classe `TemaParadizo` deixe de implementar `Plugin` e passe a implementar a nova interface `AoRenderizarHTML`. O método `aposRenderizacao` deve ser mantido inalterado. Já o método `aposGeracao` deve ser removido:

```
// br.com.paradizo.tema.TemaParadizo

~~public class TemaParadizo implements Plugin {~~
public class TemaParadizo implements AoRenderizarHTML {

  // código omitido...

  ~~@Override~~
  ~~public void aposGeracao(Ebook ebook) {~~
```

```
  }-

}
```

Ainda no projeto `tema-paradizo`, o arquivo de configuração do *service provider* deve ser renomeado para a nova SPI:

- de `META-INF/services/cotuba.plugin.Plugin`
- para `META-INF/services/cotuba.plugin.AoRenderizarHTML`

O conteúdo do arquivo deve ser mantido, apontando para a classe `br.com.paradizo.tema.TemaParadizo`. Após gerar PDFs ou EPUBs, a equipe da Paradizo deve ver os temas aplicados no ebook!

Uma interface separada para o hook de geração

A interface `AoFinalizarGeracao` deve ser criada no pacote `cotuba.plugin` do projeto `cotuba-cli`. O método `aposGeracao` deve ser definido na nova interface:

```
// cotuba.plugin.AoFinalizarGeracao

package cotuba.plugin;

import cotuba.domain.Ebook;

public interface AoFinalizarGeracao {

  void aposGeracao(Ebook ebook);

}
```

O método estático `gerou` deve ser copiado da interface `Plugin` para a nova interface `AoFinalizarGeracao`. O `ServiceLoader` deve receber a nova interface em seu método `load`:

```
// cotuba.plugin.AoFinalizarGeracao

public interface AoFinalizarGeracao {
```

```
  void aposGeracao(Ebook ebook);

  // inserido
  static void gerou(Ebook ebook) {
    ServiceLoader.load(AoFinalizarGeracao.class) // modificado!
      .forEach(plugin -> plugin.aposGeracao(ebook));
  }

}
```

O método estático `gerou` da interface `AoFinalizarGeracao` deve ser usado no final do método `executa` da classe `Cotuba`:

```
// cotuba.application.Cotuba

public class Cotuba {

  public void executa(ParametrosCotuba parametros) {

    // código omitido...

    Plugin.gerou(ebook);
    AoFinalizarGeracao.gerou(ebook);

  }

}
```

A interface `Plugin`, do pacote `cotuba.plugin`, não está sendo usada mais e pode ser removida!

```
cotuba.plugin.Plugin
```

No projeto `estatisticas-ebook`, o time da Cognitio deve obter a nova versão de `cotuba-cli` e fazer com que a classe `CalculadoraDeEstatisticas` passe a implementar `AoFinalizarGeracao`, removendo o método `aposRenderizacao`, mas mantendo o método `aposGeracao`:

```
// br.com.cognitio.estatisticas.CalculadoraDeEstatisticas
```

```
public class CalculadoraDeEstatisticas
    implements AoFinalizarGeracao {

  @Override
  public String aposRenderizacao(String html) {
  --return html;
  }

  // código omitido...

}
```

O nome do arquivo de configuração do *service provider* deve ser atualizado para a nova SPI `AoFinalizarGeracao`, no projeto `estatisticas-ebook`:

- de `META-INF/services/cotuba.plugin.Plugin`
- para `META-INF/services/cotuba.plugin.AoFinalizarGeracao`

O conteúdo do arquivo deve ser mantido como `br.com.cognitio.estatisticas.CalculadoraDeEstatisticas`.

Pronto! Em vez de uma interface em comum, temos interfaces para cada uso específico.

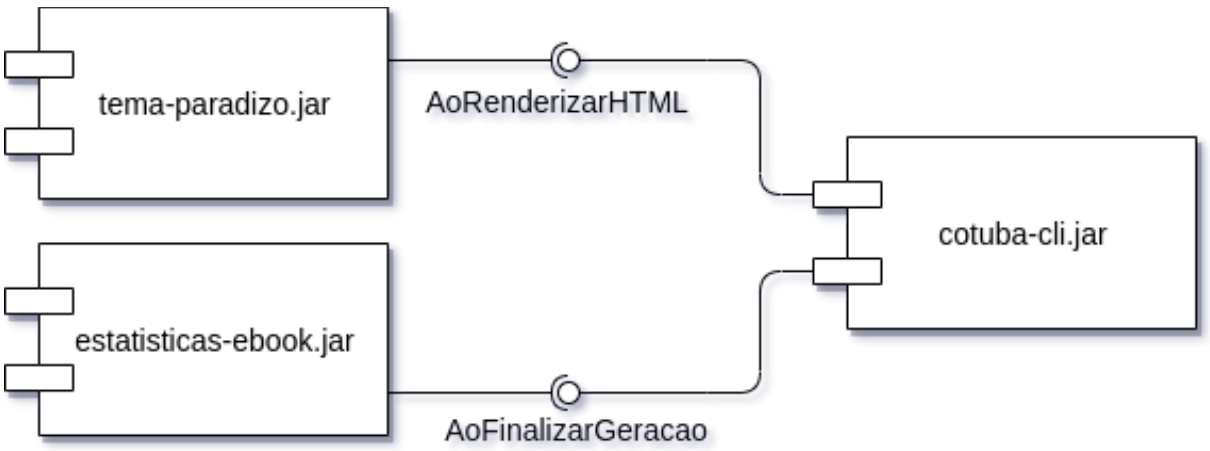

Figura 8.1: Interfaces segregadas no Cotuba

A exibição das estatísticas da Cognitio ao gerar um ebook deve continuar funcionando!

8.2 Separando classes, não apenas interfaces

Considere que temos a seguinte classe:

```
public class NotaFiscal {

  private Cliente cliente;

  private Endereco entrega;
  private Endereco cobranca;

  private List<Item> itens = new ArrayList<>();

  private List<Desconto> descontos = new ArrayList<>();
  private FormaDePagamento pagamento;
  private BigDecimal valorTotal;

  // getters...

  // restante do código...
}
```

> Esse exemplo é inspirado no livro *OO e SOLID para Ninjas* (ANICHE, 2015).

A classe `NotaFiscal` do código anterior possui vários atributos relacionados à emissão de notas fiscais. Vamos considerar que a complexidade da classe é justificada pela dificuldade do problema que está sendo resolvido.

Uma `NotaFiscal` é passada como parâmetro para o método `calcula` da classe `CalculadoraDeImposto`:

```
public class CalculadoraDeImposto {
```

```
  private static final BigDecimal VALOR_LIMITE =
      new BigDecimal(1000);
  private static final BigDecimal TAXA_MAIOR =
      new BigDecimal("0.02");
  private static final BigDecimal TAXA_MENOR =
      new BigDecimal("0.01");

  // nota fiscal usada aqui
  public BigDecimal calcula(NotaFiscal nota) {
    BigDecimal total = BigDecimal.ZERO;

    for (Item item : nota.getItens()) { // e também aqui

      if (item.getValor().compareTo(VALOR_LIMITE) > 0) {
        total = total.add(item.getValor().multiply(TAXA_MAIOR));

      } else {
        total = total.add(item.getValor().multiply(TAXA_MENOR));
      }

    }

    return total;
  }
}
```

Perceba que a classe `CalculadoraDeImposto` usa apenas o método `getItens` da `NotaFiscal`. Para o cálculo de impostos, não precisamos de outras informações da nota fiscal.

Os atributos completos da nota fiscal seriam necessários para uma classe que gera uma representação em PDF, para um DANFe (Documento Auxiliar da NF-e), ou em XML.

Já para uma classe envolvida com relacionamento com o cliente (CRM) seria interessante ter apenas os dados do cliente, o valor total e os endereços de entrega e cobrança.

Ou seja, dependendo de quem usa a `NotaFiscal`, precisaríamos de acesso a diferentes conjuntos de atributos. Deixar o acesso a todos

os atributos pode ser ruim em termos de design de código. Poderíamos trabalhar com dados de cliente ou endereços dentro de uma classe responsável por cálculos financeiros, como `CalculadoraDeImposto`, ou misturar lógica de CRM com emissão de DANFe. O que nos impediria, além do bom senso?

Uma interface específica para quem usa

No caso da `CalculadoraDeImposto`, poderíamos separar apenas aquilo que desejamos da classe `NotaFiscal` por meio de uma interface bem enxuta. Nesse caso específico, estamos apenas interessados nos itens da nota:

```
public interface Tributavel {
  List<Item> itensASeremTributados();
}
```

A interface `Tributavel` seria implementada por `NotaFiscal`:

```
public class NotaFiscal implements Tributavel {

  // código omitido...

  public List<Item> itensASeremTributados() {
    return itens;
  }

}
```

Na classe `CalculadoraDeImposto`, trocaríamos o uso de toda `NotaFiscal` pela interface `Tributavel`:

```
public class CalculadoraDeImposto {

  // código omitido...

  // modificado
  public BigDecimal calcula(Tributavel tributavel) {
    BigDecimal total = BigDecimal.ZERO;
```

```
    // modificado
    for (Item item : tributavel.itensASeremTributados()) {
      // código omitido...
    }

  }

}
```

Não teríamos mais acesso a todas as informações da nota fiscal. Apenas às informações necessárias.

> *"Se você tiver uma classe que tenha vários clientes, em vez de carregar a classe com todos os métodos de que os clientes precisam, crie interfaces específicas para cada cliente e implemente-as na classe."*
>
> Uncle Bob, no artigo *Design Principles and Design Patterns* (MARTIN, 2000)

8.3 Interfaces específicas para o cliente no Cotuba

No método `aposGeracao` do plugin `AoFinalizarGeracao`, passamos o `Ebook` que acabou de ser gerado. Há um problema: o `Ebook` possui *setters*, que poderiam ser usados pelos *service providers* para alterar informações do ebook.

Um *service provider* bem-intencionado, como a `CalculadoraDeEstatisticas` da empresa Cognitio, não faria nenhuma alteração. Mas uma implementação maliciosa do plugin `AoFinalizarGeracao` poderia remover ou trocar os capítulos, mudar o formato do ebook e/ou o arquivo de saída:

```
public class ServiceProviderMalicioso
    implements AoFinalizarGeracao {

  @Override
  public void aposGeracao(Ebook ebook) {

    ebook.setArquivoDeSaida(null); // ih!
    ebook.setCapitulos(null); // oxi!
    ebook.setFormato(null); // eita!

  }
}
```

Uma solução é criar uma interface para a classe `Ebook` específica para os plugins, que seria somente para leitura. Ou seja, teria apenas os *getters* do formato do ebook, do arquivo de saída e dos capítulos.

Mas o problema ainda persistiria: não podemos expor os setters de `Capitulo`. Um outro *service provider* malicioso pode explorar um *setter* para modificar o conteúdo HTML do capítulo:

```
public class OutroServiceProviderMalicioso
    implements AoFinalizarGeracao {

  @Override
  public void aposGeracao(Ebook ebook) {

    for (Capitulo capitulo : ebook.getCapitulos()) {

      capitulo.setTitulo(null); // oh!
      capitulo.setConteudoHTML(null); // vixe!

    }

  }

}
```

A ideia é criar uma interface específica para os plugins para o capítulo, também só para leitura.

> Em um capítulo posterior veremos uma solução diferente.

Protegendo o domain model com interfaces

Crie uma interface `CapituloSoParaLeitura` no pacote `cotuba.plugin`, baseada na classe `Capitulo` do pacote `cotuba.domain`, mas contendo apenas os getters:

```
// cotuba.plugin.CapituloSoParaLeitura

package cotuba.plugin;

public interface CapituloSoParaLeitura {

  String getTitulo();
```

```
  String getConteudoHTML();

}
```

A classe `Capitulo` do pacote `cotuba.domain` deve implementar a nova interface:

```
// cotuba.domain.Capitulo

import cotuba.plugin.CapituloSoParaLeitura; // inserido

// modificado
public class Capitulo implements CapituloSoParaLeitura {

  // código omitido...

}
```

Os métodos definidos na interface já estão implementados pela classe `Capitulo`. Baseando-se em `Ebook` do pacote `cotuba.domain`, crie uma interface `EbookSoParaLeitura` apenas com os getters, no pacote `cotuba.plugin`:

```
// cotuba.plugin.EbookSoParaLeitura

import cotuba.domain.FormatoEbook;
import java.nio.file.Path;
import java.util.List;

package cotuba.plugin;

public interface EbookSoParaLeitura {

  FormatoEbook getFormato();

  Path getArquivoDeSaida();

  List<? extends CapituloSoParaLeitura> getCapitulos();

}
```

Para podermos retornar uma lista com qualquer subtipo da interface `CapituloSoParaLeitura`, incluindo a classe `Capitulo`, devemos usar o *bounded wildcard* `List<? extends CapituloSoParaLeitura>`.

A nova interface `EbookSoParaLeitura` deve ser implementada pela classe `Ebook` do pacote `cotuba.domain`:

```
// cotuba.domain.Ebook

import cotuba.plugin.EbookSoParaLeitura; // inserido

// modificado
public class Ebook implements EbookSoParaLeitura {

  // código omitido...

}
```

Na interface `AoFinalizarGeracao`, deve ser usada a interface `EbookSoParaLeitura`:

```
// cotuba.plugin.AoFinalizarGeracao

public interface AoFinalizarGeracao {

  void aposGeracao(Ebook ebook);
  void aposGeracao(EbookSoParaLeitura ebook);

  static void gerou(Ebook ebook) {
  static void gerou(EbookSoParaLeitura ebook) {
    ServiceLoader.load(AoFinalizarGeracao.class)
      .forEach(plugin -> plugin.aposGeracao(ebook));
  }

}
```

No projeto `estatisticas-ebook`, o time da Cognitio deve corrigir a classe `CalculadoraDeEstatisticas` para que sejam usadas as novas interfaces que só possuem getters:

```java
// br.com.cognitio.estatisticas.CalculadoraDeEstatisticas

import cotuba.domain.Capitulo;
import cotuba.plugin.CapituloSoParaLeitura;

import cotuba.domain.Ebook;
import cotuba.plugin.EbookSoParaLeitura;

public class CalculadoraDeEstatisticas
    implements AoFinalizarGeracao {

  // modificado
  @Override
  public void aposGeracao(EbookSoParaLeitura ebook) {

    ContagemPalavras contagemDePalavras = new ContagemPalavras();

    // modificado
    for (CapituloSoParaLeitura capitulo : ebook.getCapitulos()) {

      // código omitido...

    }

    // código omitido...

  }

}
```

Pronto! A geração de ebooks e as estatísticas da Cognitio devem funcionar normalmente. E agora não há chance de um plugin modificar o ebook!

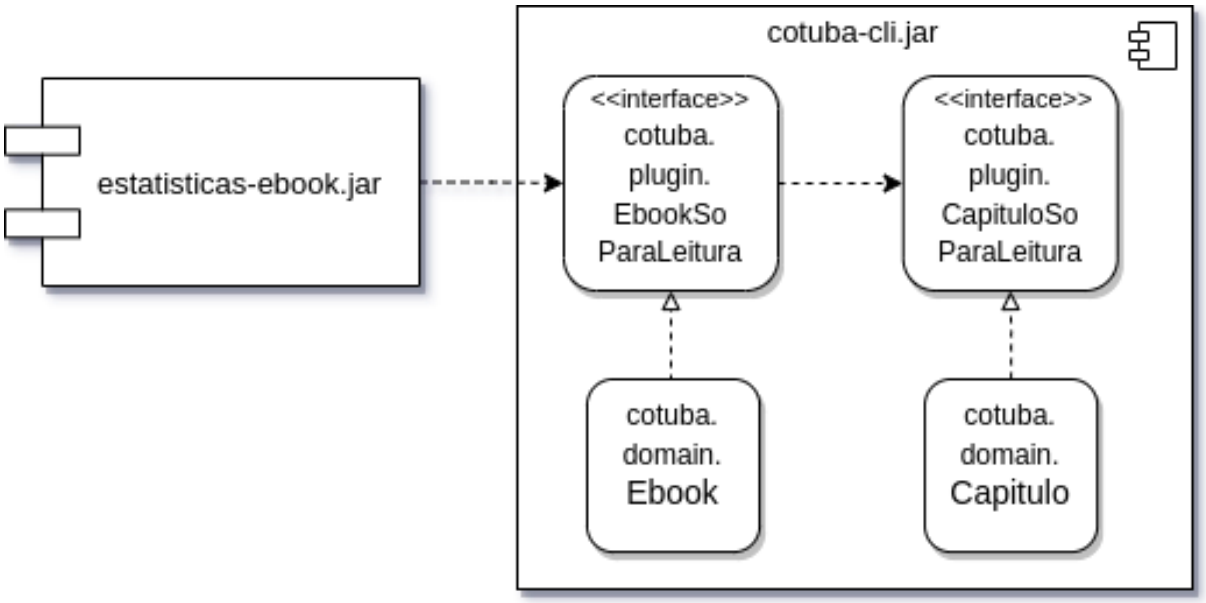

Figura 8.2: Interfaces de domínio segregadas

Para saber mais: Atitude limitante vs. permissiva

No artigo *Software Development Attitude* (FOWLER, 2004), Martin Fowler argumenta que há duas atitudes quando discutimos sobre linguagens, metodologias, ferramentas e design de código:

- *Directing attitude*: uma atitude limitante, controladora, direcionadora, restritiva. É uma mentalidade preocupada em prevenir desenvolvedores ruins de causar danos. Leva a designs robustos, mas que limitam as possibilidades.
- *Enabling attitude*: uma atitude permissiva, habilitante, tolerante, capacitadora. É uma mentalidade que dá liberdade aos desenvolvedores, sem prevenir que façam coisas ruins. Leva a designs flexíveis, mas que podem ser usados de maneira errada.

Ao protegermos o domain model do Cotuba estamos tomando uma atitude limitante.

8.4 Contraponto: críticas ao ISP

Dan North, no artigo *CUPID - the back story* (NORTH, 2021), conta a história que o ISP foi cunhado por Uncle Bob enquanto quebrava uma classe massiva de um software de impressão da Xerox em classes menores. Por isso, North diz que o ISP não é um princípio, mas um pattern: uma estratégia que funciona em um determinado contexto e traz vantagens e desvantagens. O autor argumenta ainda que não haveria a necessidade de quebrar uma classe grande e emaranhada se o código já fosse composto por classes menores.

David Copeland diz, no livro *SOLID is not solid* (COPELAND, 2019), que o ISP só faz sentido em linguagens com tipagem estática como Java e Scala. Copeland diz ainda que a necessidade do ISP só surge quando há classes pouco coesas e que o ISP levado ao extremo leva a uma proliferação de interfaces de apenas um método. O autor conclui afirmando que o ISP é uma reformulação complicada sobre a necessidade de coesão.

Ted Kaminski argumenta, em seu artigo *Deconstructing SOLID design principles* (KAMINSKI, 2019), o ISP é vago e abre espaço para algumas interpretações. Uma das interpretações é que interfaces pouco coesas são tão destrutivas quanto objetos pouco coesos e que múltiplas interfaces são melhores que uma megainterface única porque controlam e limitam dependências públicas. Uma outra interpretação está ligada à evolução de plugins e APIs que precisam dar suporte a código legado e que precisam manter versões anteriores de suas interfaces públicas. Essa segunda interpretação, para Kaminski, não é um princípio para ser adotado inicialmente, mas uma ferramenta para lidar com as fronteiras de um sistema.

8.5 O que aprendemos?

Neste capítulo, quebramos a interface `Plugin` em duas interfaces mais coesas: `AoRenderizarHTML` e `AoFinalizarGeracao`. Dessa forma, os *service providers* `TemaParadizo` e `CalculadorDeEstatisticas` não precisam depender de métodos que não usam. Em seguida, fizemos versões só para leitura do domain model do Cotuba, para que código malicioso não explorasse o plugin chamado ao final da geração de um ebook. Com isso, atendemos ao ISP, fazendo com que diminuindo a "superfície" exposta pelas classes e interfaces do Cotuba. E estudamos todas as letras do acrônimo SOLID!

No próximo capítulo, estudaremos como imutabilidade e encapsulamento podem ajudar podem ajudar a minimizar alguns problemas do design do código do Cotuba.

O código deste capítulo pode ser encontrado em: https://github.com/alexandreaquiles/solid-na-pratica/tree/cap8-interface-segregation-principle

Figura 8.3: Vídeo do capítulo

CAPÍTULO 9

Um pouco de imutabilidade e encapsulamento

No capítulo anterior, criamos as interfaces `EbookSoParaLeitura` e `CapituloSoParaLeitura` no pacote `cotuba.plugin`, que continham apenas *getters* dos atributos, para que os plugins não tivessem acesso aos *setters* das classes do domain model do Cotuba. A solução atende ao ISP, já que não fornece aos clientes métodos desnecessários. Mas será que a solução atende ao DIP?

As classes `Ebook` e `Capitulo`, do pacote `cotuba.domain`, implementam interfaces do pacote `cotuba.plugin`. Se classificarmos o código relacionado aos plugins como de baixo nível, estaríamos quebrando o DIP. Classes relacionadas ao negócio estariam implementando abstrações não relacionadas ao negócio. Ao atendermos ao ISP, acabamos quebrando o DIP.

A motivação para criamos as interfaces `Ebook` e `Capitulo` do pacote `cotuba.plugin` foi evitar que os plugins tivessem acessos aos setters. Mas e se, nas próprias classes do pacote `cotuba.domain`, não tivéssemos esses setters? Os plugins não poderiam chamar métodos que não existem. Mas é possível criar uma classe sem setters? Sim! Estaríamos caminhando na direção da imutabilidade.

9.1 Em direção à imutabilidade

Vamos considerar uma classe `ContaCorrente`, que possui os atributos `saldo`, do tipo `BigDecimal` e `movimentacoes`, uma `List<Movimentacao>`. Os getters e setters são definidos para ambos os atributos. Há ainda os métodos `deposita`, `saca` e `adicionaMovimentacao`:

```
public class ContaCorrente {

  private BigDecimal saldo;
  private List<Movimentacao> movimentacoes;

  public BigDecimal getSaldo() {
    return saldo;
  }

  public List<Movimentacao> getMovimentacoes() {
    return movimentacoes;
  }

  public void setSaldo(BigDecimal saldo) {
    this.saldo = saldo;
  }

  public void setMovimentacoes(List<Movimentacao> movimentacoes) {
    this.movimentacoes = movimentacoes;
  }

  public void deposita(BigDecimal valor) {
    saldo = saldo.add(valor);
  }

  public void saca(BigDecimal valor) {
    saldo = saldo.subtract(valor);
  }
```

```
  public void adicionaMovimentacao(Movimentacao movimentacao) {
    movimentacoes.add(movimentacao);
  }

}
```

Essa *não* é uma classe imutável. Uma classe imutável é aquela cujos objetos, depois de instanciados, não mudam os valores de seus atributos. Certamente, setters não devem ser definidos quando nos preocupamos com imutabilidade.

```
public class ContaCorrente {

  // código omitido...

  public void setSaldo(BigDecimal saldo) {
    return this.saldo = saldo;
  }

  public void setMovimentacoes(List<Movimentacao> movimentacoes) {
    return this.movimentacoes = movimentacoes;
  }

   // código omitido...

}
```

Ainda assim, uma classe que não tem setters não necessariamente será imutável. Para que seja, é preciso que *nenhum outro* método mude o estado de um objeto dessa classe, isto é, que os valores dos seus atributos não sejam modificados.

Para garantir que não haverá mudança de estado, podemos definir os atributos como `final`. Qualquer tentativa de mudança de um valor de um atributo fora da inicialização do objeto ocasionará um erro de compilação:

```
public class ContaCorrente {

  private final BigDecimal saldo; // modificado
```

```
  private final List<Movimentacao> movimentacoes; // modificado

  // código omitido...

  public void deposita(BigDecimal valor) {
    saldo = saldo.add(valor); // error: cannot assign a value to final
variable
  }

  public void saca(BigDecimal valor) {
    saldo = saldo.subtract(valor); // error: cannot assign a value to
final variable
  }

  // código omitido...

}
```

Com atributos `final`, não podemos ter setters nem métodos que mudam seus valores. Mas, então, como fazer atribuições? Por meio de construtores! Os atributos têm seus valores iniciais informados na criação do objeto. Uma vez definidos, esses valores não podem ser modificados:

```
public class ContaCorrente {

  private final BigDecimal saldo;
  private final List<Movimentacao> movimentacoes;

  public ContaCorrente(BigDecimal saldo,
      List<Movimentacao> movimentacoes) {
    this.saldo = saldo;
    this.movimentacoes = movimentacoes;
  }

  // código omitido...

}
```

Se for necessário, podemos definir mais construtores, em que informamos apenas parte dos atributos. Mas e os métodos que fazem mutações nos atributos? Passam a retornar novos objetos:

```
public class ContaCorrente {

  // código omitido...

  public ContaCorrente deposita(BigDecimal valor) {
    BigDecimal novoSaldo = this.saldo.add(valor);
    return new ContaCorrente(novoSaldo, this.movimentacoes);
  }

  public ContaCorrente saca(BigDecimal valor) {
    BigDecimal novoSaldo = this.saldo.subtract(valor);
    return new ContaCorrente(novoSaldo, this.movimentacoes);
  }

  // código omitido...

}
```

Ainda não temos uma classe imutável já que seria possível criar uma classe filha que se comportasse como se houvesse mudança de estado. Porém, é possível usar `final` na classe, sinalizando que não poderão existir subclasses:

```
public final class ContaCorrente {

  // código omitido...

}
```

Há ainda uma questão importante para chegarmos à imutabilidade de uma classe: se houver composição com objetos mutáveis, seria possível que outras classes mudassem seus estados. Bastaria que obtivessem referências a esses objetos via getters ou de alguma outra maneira.

Não há problemas na composição com `BigDecimal`, pois essa classe já é imutável! Mas há essa questão da composição com objetos mutáveis acontece no método `getMovimentacoes` de `ContaCorrente`. Uma `List<Movimentacao>`, que é mutável, é compartilhada com outros objetos. Outros objetos podem adicionar, remover ou trocar valores dessa lista, mudando o estado da `ContaCorrente` indiretamente.

Para resolver isso, é preciso fazer uma **cópia defensiva** dos objetos mutáveis usados por uma classe que pretende ser imutável. No caso de uma `List`, há o método `unmodifiableList` da classe utilitária `Collections`. Esse método retorna uma cópia *só para leitura* da lista original:

```
public final class ContaCorrente {

  private final BigDecimal saldo;
  private final List<Movimentacao> movimentacoes;

  public ContaCorrente(BigDecimal saldo,
      List<Movimentacao> movimentacoes) {
    this.saldo = saldo;
    this.movimentacoes =
        Collections.unmodifiableList(movimentacoes);
  }

    public BigDecimal getSaldo() {
    return saldo;
  }

  public List<Movimentacao> getMovimentacoes() {
    return movimentacoes;
  }

  // código omitido...

}
```

Se um objeto obtiver a lista de movimentações por meio do método `getMovimentacoes`, não será possível adicionar, remover ou trocar

valores dessa lista. Qualquer chamada aos métodos que mudariam uma `unmodifiableList` faz com que seja lançada uma `UnsupportedOperationException`.

É interessante notar que a documentação do Java evita afirmar que uma *unmodifiable collection* é imutável, porque os elementos contidos na coleção podem ser mutáveis. Por isso, é usado o termo "não modificável".

Em versões mais recentes do Java, há outras maneiras de ter uma versão não modificável de uma coleção:

```
// desde o Java 1.2
this.movimentacoes = Collections.unmodifiableList(movimentacoes);

// desde o Java 9
this.movimentacoes = List.of(movimentacoes.toArray());

// desde o Java 16
this.movimentacoes = movimentacoes.stream().toList();
```

Vale lembrar um detalhe do capítulo sobre LSP: as *unmodifiable collections* do Java implementam as interfaces mutáveis, como `List` e `Set`, mas lançam exceções quando métodos mutáveis são invocados. Um código que recebe uma `List`, por exemplo, pode ter uma surpresa indesejada ao tentar adicionar um elemento. Há uma quebra do LSP.

Classes imutáveis na plataforma Java

Boa parte das classes da biblioteca padrão do Java são imutáveis. Exemplos:

- `String`
- classes wrappers de tipos primitivos como `Integer` e `Double`
- `BigDecimal` e `BigInteger`
- classes da API `java.time` como `LocalDate` e `LocalTime`

Classes mutáveis são comuns na API do Java, como:

- `Calendar` e `Date`
- implementações padrão da API de Collections como `ArrayList`, `HashSet` e `TreeMap`.

Imutabilidade

Um dos motes de Joshua Bloch no livro *Effective Java* (BLOCH, 2001) é: *"Item 13: Favoreça a imutabilidade."* O autor, ainda no mesmo livro, lista cinco "regras" para tornar uma classe imutável:

- Não forneça nenhum método que modifique o estado do objeto.
- Assegure que a classe não possa ser estendida.
- Defina todos os atributos como `final`.
- Faça com que todos os atributos sejam privados.
- Assegure acesso exclusivo a qualquer composição com objetos mutáveis.

Algumas vantagens de objetos imutáveis citadas por Bloch no mesmo livro:

- São simples: só há um estado possível.
- São *thread-safe*: como não há mudança de estado, não há problemas no acesso concorrente aos mesmos objetos.
- Podem ser compartilhados e reutilizados livremente.

A grande desvantagem dos objetos imutáveis é no uso de memória: para cada valor distinto, é preciso um novo objeto.

> *"Classes devem ser imutáveis ao menos que haja uma boa razão para torná-las mutáveis. (...) Se uma classe não puder ser imutável, limite sua mutabilidade o máximo possível."*
>
> Joshua Bloch, no livro *Effective Java* (BLOCH, 2001)

Mais pragmatismo, menos fundamentalismo

Depois de ler as dicas de Joshua Bloch quanto à imutabilidade, dá aquela vontade de apagar tudo o que já implementamos, removendo setters, colocando atributos e classes como `final`, não é mesmo? Tenhamos calma!

Na plataforma Java, é muito difícil criar código 100% imutável. As especificações e as bibliotecas que as implementam, em geral, requerem JavaBeans. É assim com JPA/Hibernate, com o JSF, com o JAX-B, com o JavaFX, entre outros. E um JavaBean precisa de getters, setters e de um construtor sem parâmetros.

Poderíamos deixar nossos objetos de alto nível como imutáveis, criando JavaBeans específicos para cada tecnologia da plataforma Java. Quando necessário, faríamos a conversão de/para os objetos imutáveis. Porém, quanto mais intermediários no código, mais complexidade.

Uma abordagem mais pragmática e menos idealista é trabalhar com objetos mutáveis, mesmo que os objetos de negócio acabem, aos poucos, sendo tomados por essa mutabilidade.

Escolher entre idealismo e pragmatismo é uma decisão que deve ser tomada de acordo com o contexto do projeto, da equipe e da empresa.

9.2 Um domain model imutável

Para iniciar os passos na direção da imutabilidade, é importante remover os setters. Primeiro, para a classe `Ebook` do pacote `cotuba.domain`:

```
// cotuba.domain.Ebook

public class Ebook implements EbookSoParaLeitura {

  // código omitido...

  public void setFormato(FormatoEbook formato) {
    this.formato = formato;
  }

  public void setArquivoDeSaida(Path arquivoDeSaida) {
    this.arquivoDeSaida = arquivoDeSaida;
  }

  public void setCapitulos(List<Capitulo> capitulos) {
    this.capitulos = capitulos;
  }

  }
```

O mesmo deve ser feito para o `Capitulo`:

```
// cotuba.domain.Capitulo

public class Capitulo implements CapituloSoParaLeitura {

  // código omitido...

  public void setTitulo(String titulo) {
    this.titulo = titulo;
  }
```

```
  public void setConteudoHTML(String conteudoHTML) {
    this.conteudoHTML = conteudoHTML;
  }

}
```

Ao remover os setters, ocorrerão erros de compilação em diversas classes. Para impedir que subclasses mutáveis sejam definidas, deve ser definido o modificador `final` em `Ebook`:

```
// cotuba.domain.Ebook

// modificado
public final class Ebook implements EbookSoParaLeitura {

  // código omitido...

}
```

Algo semelhante deve ser feito para `Capitulo`:

```
// cotuba.domain.Capitulo

// modificado
public final class Capitulo implements CapituloSoParaLeitura {

  // código omitido...

}
```

Para caminhar mais um passo na direção da imutabilidade, os atributos das classes devem ser `final`, recebendo os valores iniciais em um construtor.

> Dica: a IDE pode ajudar!

Em `Ebook`:

```
// cotuba.domain.Ebook

public final class Ebook implements EbookSoParaLeitura {

  private final FormatoEbook formato; // modificado
  private final Path arquivoDeSaida; // modificado
  private final List<Capitulo> capitulos; // modificado

  // inserido
  public Ebook(FormatoEbook formato, Path arquivoDeSaida,
      List<Capitulo> capitulos) {
    this.formato = formato;
    this.arquivoDeSaida = arquivoDeSaida;
    this.capitulos = capitulos;
  }

  // código omitido...

}
```

Em `Capitulo`:

```
// cotuba.domain.Capitulo

public final class Capitulo implements CapituloSoParaLeitura {

  private final String titulo; // modificado
  private final String conteudoHTML; // modificado

  // inserido
  public Capitulo(String titulo, String conteudoHTML) {
    this.titulo = titulo;
    this.conteudoHTML = conteudoHTML;
  }

  // código omitido...

}
```

Na classe `Ebook` do pacote `cotuba.domain`, não pode ser mantida a composição com uma lista mutável. Por isso, a `List<Capitulo>` de

`Ebook` deve ser passada para o método estático `unmodifiableList` da classe utilitária `Collections`:

```
// cotuba.domain.Ebook

import java.util.Collections; // inserido

public final class Ebook implements EbookSoParaLeitura {

  private final FormatoEbook formato;
  private final Path arquivoDeSaida;
  private final List<Capitulo> capitulos;

  public Ebook(FormatoEbook formato, Path arquivoDeSaida,
      List<Capitulo> capitulos) {
    this.formato = formato;
    this.arquivoDeSaida = arquivoDeSaida;

    // modificado
    this.capitulos = Collections.unmodifiableList(capitulos);
  }

  // código omitido...

}
```

Os erros de compilação ainda devem continuar. Como corrigi-los?

Corrigindo a classe Cotuba

Os erros na classe `Cotuba` acontecem porque é criado um `Ebook` com o construtor padrão, sem parâmetros, e, em seguida, são chamados os setters.

```
// cotuba.application.Cotuba

Ebook ebook = new Ebook(); // error: The constructor is undefined
ebook.setFormato(formato); // error:  The method is undefined for the type
ebook.setArquivoDeSaida(arquivoDeSaida); // error: The method is undefined
for the type
```

```
ebook.setCapitulos(capitulos); // error: The method is undefined for the type
```

Nada disso existe mais! A correção, nesse caso, é simples: basta que seja usado o novo construtor da classe `Ebook`.

```
// cotuba.application.Cotuba

Ebook ebook = new Ebook(formato, arquivoDeSaida, capitulos);
```

Corrigindo o hook de renderização

O método estático `renderizou` da interface `AoRenderizarHTML` usa um setter para atualizar o conteúdo do capítulo com o HTML retornado pelas implementações da SPI. Como a classe `Capitulo` agora é imutável, esse trecho de `AoRenderizarHTML` apresentará um erro de compilação:

```
// cotuba.plugin.AoRenderizarHTML

public interface AoRenderizarHTML {

  String aposRenderizacao(String html);

  static void renderizou(Capitulo capitulo) {
    ServiceLoader.load(AoRenderizarHTML.class)
      .forEach(plugin -> {
        String html = capitulo.getConteudoHTML();
        String htmlModificado = plugin.aposRenderizacao(html);
        capitulo.setConteudoHTML(htmlModificado); // error: The method is undefined for the type
      });
  }

}
```

Devemos remover a chamada ao setter. Mas como atualizar o HTML de um capítulo se não podemos o conteúdo não pode ser modificado? Há diversos caminhos a seguir. No que vamos tomar, é necessário modificar a assinatura do método estático `renderizou`

para, em vez de receber um capítulo, receber um HTML e retornar uma `String` com o HTML modificado:

```
// cotuba.plugin.AoRenderizarHTML

public interface AoRenderizarHTML {

  String aposRenderizacao(String html);

  static void renderizou(Capitulo capitulo) {
  static String renderizou(String html) {

    ServiceLoader.load(AoRenderizarHTML.class)
      .forEach(plugin -> {
        String html = capitulo.getConteudoHTML();

        String htmlModificado = plugin.aposRenderizacao(html);

        // O que fazer com htmlModificado?

        capitulo.setConteudoHTML(htmlModificado);
      });
  }

}
```

Há um problema: o `forEach` usado para percorrer os *service providers* da SPI `AoRenderizarHTML` é um método sem retorno. Por isso, não podemos retornar o HTML modificado. Uma ideia é usar um *for-each* padrão, passando o HTML modificado por cada *service provider* para o próximo:

```
// cotuba.plugin.AoRenderizarHTML

public interface AoRenderizarHTML {

  String aposRenderizacao(String html);

  static String renderizou(String html) {
    // modificado
```

```
    String htmlModificado = html;
    for (AoRenderizarHTML plugin :
        ServiceLoader.load(AoRenderizarHTML.class)) {
      htmlModificado = plugin.aposRenderizacao(htmlModificado);
    }
    return htmlModificado;
  }

}
```

Dessa forma, cada HTML modificado pelas implementações da SPI `AoRenderizarHTML` é passado para o seguinte, acumulando as transformações feitas pelos *service providers*. Se não houver nenhum provider para a SPI, o HTML original é retornado.

Na classe `RenderizadorMDParaHTML`, é necessário adequar o código para usar a nova assinatura do método estático `renderizou` da SPI de renderização:

```
// cotuba.md.RenderizadorMDParaHTML

String html = renderer.render(document);
capitulo.setConteudoHTML(html);

AoRenderizarHTML.renderizou(capitulo);
String htmlModificado = AoRenderizarHTML.renderizou(html);

capitulo.setConteudoHTML(htmlModificado); // error: The method is
undefined for the type
```

Uma questão importante é que não poderemos usar o setter do capítulo para definir o HTML modificado. O que fazer?

Corrigindo o renderizador

O problema na classe `RenderizadorMDParaHTML` é o mais complexo. Há erros de compilação nas chamadas aos setters e ao construtor

padrão de `Capitulo`:

```
// cotuba.md.RenderizadorMDParaHTML

public class RenderizadorMDParaHTML {

  public List<Capitulo> renderiza(Path diretorioDosMD) {
    // código omitido...
    Capitulo capitulo = new Capitulo(); // error: The constructor is
undefined
    // código omitido...
  }

  private Node parseDoMD(Path arquivoMD, Capitulo capitulo) {
    // código omitido...
    capitulo.setTitulo(tituloDoCapitulo); // error: The method is
undefined for the type
    // código omitido...
  }

  private void renderizaParaHTML(Path arquivoMD,
      Capitulo capitulo, Node document) {
    // código omitido...
    capitulo.setConteudoHTML(htmlModificado); // error: The method is
undefined for the type
  }
}
```

A questão é que não basta agruparmos tudo em uma chamada ao construtor definido na classe `Capitulo`, que recebe o título e o conteúdo HTML. No momento da criação do objeto com o construtor padrão, ainda não temos todas as informações necessárias. Precisamos processar os arquivos MD para descobrir o título de cada capítulo. Para ter o conteúdo, temos que renderizar cada MD para HTML.

Há várias soluções para esse caso específico, em que queremos construir um objeto aos poucos. A maneira mais simples seria armazenar o título e, depois, o conteúdo HTML em variáveis para,

só então, usar o construtor de `Capitulo`. Mas podemos usar uma solução mais sofisticada.

9.3 Design Pattern: Builder

A criação de alguns objetos é complexa. Podem ser necessárias muitas informações e/ou muitos passos no seu processo de instanciação. Uma nota fiscal, por exemplo, contém informações como:

- data de emissão
- nome completo do cliente
- endereço do cliente
- lista de produtos

Um construtor de uma classe `NotaFiscal` receberia muitos parâmetros.

```
public final class NotaFiscal {

  // atributos omitidos...

  public NotaFiscal(LocalDate data, String nomeCliente,
      String enderecoCliente, List<Produto> produtos) {
    // parâmetros do construtor setados nos atributos...
  }

}
```

Qual a chance de trocarmos o nome do cliente pelo endereço? Grande, não? E, provavelmente, teríamos mais atributos a serem definidos: endereço de cobrança, número de parcelas, método de pagamento etc. O construtor tende a crescer. E a dificuldade de manutenção do código aumentará.

Porém, podemos criar uma classe responsável por representar as etapas da construção do objeto. É o que chamam no livro do *Design*

Patterns (GAMMA et al., 1994) de **Builder**. Um possível uso de um Builder para a nota fiscal seria:

```
NotaFiscal nota =
  new NotaFiscalBuilder()
    .naData(2018, 9, 5)
    .paraOCliente("João da Silva")
    .doEndereco("Rua Vergueiro, 3185, 8º andar")
    .comOProduto("Livro Git e GitHub", 39.9)
    .constroi();
```

Como defini-lo? Uma implementação simples seria:

```
public class NotaFiscalBuilder {

  private LocalDate data;
  private String nomeCliente;
  private String endereco;
  private List<Produto> produtos = new ArrayList<>();

  public NotaFiscalBuilder naData(int ano, int mes, int dia) {
    this.data = LocalDate.of(ano, mes, dia);
    return this;
  }

  public NotaFiscalBuilder paraOCliente(String nome) {
    this.nomeCliente = nome;
    return this;
  }

  public NotaFiscalBuilder doEndereco(String endereco) {
    this.endereco = endereco;
    return this;
  }

  public NotaFiscalBuilder comOProduto(String nome, double preco) {
    this.produtos.add(new Produto(nome, preco));
    return this;
  }

  public NotaFiscal constroi() {
```

```
    return new NotaFiscal(data, nomeCliente, endereco, produtos);
  }

}
```

Perceba que os métodos intermediários do Builder retornam `this`, ou seja, o próprio Builder. Isso permite que as chamadas sejam encadeadas, dando maior fluência e legibilidade.

Há implementações mais elaboradas, que usam diferentes Builders para cada etapa. Assim, a construção do objeto é guiada para ser usada em uma determinada ordem, evitando estados inválidos. Além disso, o *auto-complete* das IDEs é potencializado.

> *"Use o Builder pattern quando:*
>
> - *o algoritmo para criar um objeto complexo deve ser independente das partes que compõem o objeto e como elas são montadas.*
> - *o processo de construção deve permitir diferentes representações para o objeto que é construído."*
>
> GoF (Gamma & Helm & Johnson & Vlissides) no livro *Design Patterns* (GAMMA et al., 1994)

Um Builder pode ser definido de maneira fortemente tipada para, com isso, criar uma API fluente que guia a pessoa que está programando, impedindo a construção de objetos em estado inválido e encapsulando detalhes internos. É o que mostra Gabriel Ronei, em seu artigo *Monte seu Type-Safe Builder* (RONEI, 2021).

Um Builder para os capítulos

Vamos criar uma classe `CapituloBuilder` em um novo pacote `cotuba.domain.builder`. A nova classe deve definir atributos para o título e o conteúdo HTML do capítulo, além de métodos de mutação

para cada atributo, que retornam o próprio Builder. É importante também que seja definido um método que instancia o capítulo:

```
// cotuba.domain.builder.CapituloBuilder

package cotuba.domain.builder;

import cotuba.domain.Capitulo;

public class CapituloBuilder {

  private String titulo;
  private String conteudoHTML;

  public CapituloBuilder comTitulo(String titulo) {
    this.titulo = titulo;
    return this;
  }

  public CapituloBuilder comConteudoHTML(String conteudoHTML) {
    this.conteudoHTML = conteudoHTML;
    return this;
  }

  public Capitulo constroi() {
    return new Capitulo(titulo, conteudoHTML);
  }

}
```

No método `renderiza` da classe `RenderizadorMDParaHTML`, a instanciação de `Capitulo` deve ser trocada por `CapituloBuilder`, que deve ser passado para os demais métodos. Ao final, deve ser o `Capitulo` deve ser construído a partir do Builder:

```
// cotuba.md.RenderizadorMDParaHTML

import cotuba.domain.builder.CapituloBuilder; // inserido

public class RenderizadorMDParaHTML {
```

```
  public List<Capitulo> renderiza(Path diretorioDosMD) {
    return obtemArquivosMD(diretorioDosMD).stream()
      .map(arquivoMD -> {
        Capitulo capitulo = new Capitulo();
        CapituloBuilder capituloBuilder = new CapituloBuilder();
        Node document = parseDoMD(arquivoMD, capituloBuilder);
        renderizaParaHTML(arquivoMD, capituloBuilder, document);
        return capituloBuilder.constroi();
      }).toList();
  }

}
```

Seguindo em `RenderizadorMDParaHTML`, o método `parseDoMD` deve receber o Builder e usá-lo para definir o título do capítulo, removendo o uso do setter:

```
// cotuba.md.RenderizadorMDParaHTML

// modificado
private Node parseDoMD(Path arquivoMD,
    CapituloBuilder capituloBuilder) {
  // código omitido...
  capitulo.setTitulo(tituloDoCapitulo);
  capituloBuilder.comTitulo(tituloDoCapitulo);
  // código omitido...
}
```

Ainda em `RenderizadorMDParaHTML`, no método `renderizaParaHTML`, é necessário remover o uso do setter na definição do HTML do capítulo, passando o HTML modificado pelas implementações da SPI para o Builder recebido como parâmetro:

```
// cotuba.md.RenderizadorMDParaHTML

// modificado
private void renderizaParaHTML(Path arquivoMD,
    CapituloBuilder capituloBuilder, Node document) {
  // código omitido...
  capitulo.setConteudoHTML(html);
  capituloBuilder.comConteudoHTML(html);
```

```
  // código omitido...
```

Dessa forma, as classes do domain model `Ebook` e `Capitulo` são imutáveis e a responsabilidade de construir um capítulo passo a passo foi passada para o `CapituloBuilder`.

Pronto! Com isso, todos os erros de compilação devem ter sido corrigidos. A geração de ebooks e os plugins devem funcionar normalmente!

9.4 Voltando atrás na segregação de interfaces

Muitas vezes, uma mudança nas decisões de design do nosso código faz com que certos trechos não sejam mais necessários, podendo ser simplificados.

É o caso da minimização da mutabilidade das classes `Ebook` e `Capitulo`, que fizeram com que essas classes não tenham setters. Como não há maneira de modificar os objetos do domain model por meio de seus métodos públicos, *service providers* maliciosos não poderão explorá-los. Por isso, não há mais a necessidade das versões só para leitura do nosso domain model. Portanto, as classes `EbookSoParaLeitura` e `CapituloSoParaLeitura` podem ser apagadas do pacote `cotuba.plugin`:

```
cotuba.plugin.CapituloSoParaLeitura
cotuba.plugin.EbookSoParaLeitura
```

No projeto `cotuba-cli`, devem ocorrer erros de compilação nas classes do domain model. A classe `Ebook`, do pacote `cotuba.domain`, pode deixar de implementar `EbookSoParaLeitura` e ter o import removido:

```
// cotuba.domain.Ebook

import cotuba.plugin.EbookSoParaLeitura;

public final class Ebook implements EbookSoParaLeitura {

  // código omitido...

}
```

Algo semelhante deve ser feito na classe `Capitulo`:

```
// cotuba.domain.Capitulo

import cotuba.plugin.CapituloSoParaLeitura;

public final class Capitulo implements CapituloSoParaLeitura {

  // código omitido...

}
```

Também haverá erros de compilação na interface `AoFinalizarGeracao` do pacote `cotuba.plugin`. Em vez de `EbookSoParaLeitura`, deve ser usada a classe `Ebook`:

```
// cotuba.plugin.AoFinalizarGeracao

import cotuba.domain.Ebook; // inserido

public interface AoFinalizarGeracao {

  void aposGeracao(EbookSoParaLeitura ebook);
  void aposGeracao(Ebook ebook);

  static void gerou(EbookSoParaLeitura ebook) {
  static void gerou(Ebook ebook) {
    ServiceLoader.load(AoFinalizarGeracao.class)
      .forEach(plugin -> plugin.aposGeracao(ebook));
  }

}
```

Nesse momento, todas as classes do projeto `cotuba-cli` devem compilar com sucesso.

Já no projeto `estatisticas-ebook`, a equipe da Cognitio deve enfrentar um erro de compilação na classe `CalculadoraDeEstatisticas`. Como a interface `AoFinalizarGeracao` foi alterada, o código de `CalculadoraDeEstatisticas` deve ser atualizado:

```
// br.com.cognitio.estatisticas.CalculadoraDeEstatisticas
```

```
import cotuba.plugin.CapituloSoParaLeitura;
import cotuba.domain.Capitulo;

import cotuba.plugin.EbookSoParaLeitura;
import cotuba.domain.Ebook;

public class CalculadoraDeEstatisticas
    implements AoFinalizarGeracao {

  @Override
  public void aposGeracao(EbookSoParaLeitura ebook) {
  public void aposGeracao(Ebook ebook) {

    ContagemPalavras contagemDePalavras = new ContagemPalavras();

    for (Capitulo capitulo : ebook.getCapitulos()) {
    for (CapituloSoParaLeitura capitulo : ebook.getCapitulos()) {

      // código omitido...

    }

    // código omitido...

  }

}
```

Ufa! Não devem ocorrer outros erros de compilação. Desenvolvedores de *service providers* maliciosos ou descuidados não conseguiriam modificar, por meio de sua API pública, nem os ebooks nem seus capítulos, graças à imutabilidade desses objetos!

9.5 Favorecendo imutabilidade com Records

A partir do Java 16, a linguagem Java tem uma sintaxe enxuta para declarar classes que carregam dados imutáveis: os records.

Um `record` é mais um tipo que pode ser declarado no Java, além de `class`, `interface`, `@interface` e `enum`. Um record possui um nome e "componentes", seus parâmetros, que declaram seu estado e são usados para gerar automaticamente:

- um atributo `private final`, imutável, para cada componente, com o mesmo nome e tipo;
- um método de acesso `public` para cada componente, com o mesmo nome do componente e com retorno do mesmo tipo;
- um construtor "canônico", que recebe cada componente e atribui ao atributos privados;
- implementações de `equals` e `hashcode`, que verificam cada componente;
- uma implementação de `toString` que mostra o nome de cada componente com o seu respectivo valor.

Um detalhe importante é que os métodos de acesso gerados em um record não possuem o prefixo `get`. Portanto, para um componente `saldo`, termos um método `saldo()`. Um record **não** pode:

- ser `abstract`
- declarar atributos de instância
- herdar de nenhuma classe, mas herda implicitamente de `java.lang.Record`

Fora isso, um record pode implementar interfaces, declarar membros estáticos, declarar métodos, usar modificadores de acesso e uma série de outros detalhes descritos na JEP 395: Records (https://openjdk.java.net/jeps/395).

Usando records no domain model do Cotuba

Podemos usar records nas classes imutáveis `Ebook` e `Capitulo`, simplificando enormemente o código. Não precisaremos da declaração dos atributos, dos construtores, dos getters, nem dos modificadores `final`. A classe `Capitulo`, quando transformada em record, ficaria bastante enxuta:

```
// cotuba.domain.Capitulo

public record Capitulo(String titulo, String conteudoHTML) {

}
```

Sem nada dentro! Ganhamos de graça os atributos imutáveis, os métodos de acesso, o construtor canônico e as implementações de `equals`, `hashCode` e `toString`! A classe `Ebook` se transformaria em um record um pouco mais complexo:

```
// cotuba.domain.Ebook

public record Ebook(FormatoEbook formato, Path arquivoDeSaida,
    List<Capitulo> capitulos) {

  public Ebook { // compact constructor
    capitulos = Collections.unmodifiableList(capitulos);
  }

  public boolean ultimoCapitulo(Capitulo capitulo) {
    return this.capitulos.get(this.capitulos.size() - 1)
        .equals(capitulo);
  }

}
```

Para garantir a imutabilidade da composição com `List`, usamos a sintaxe de *compact constructor*, em que não são definidos parâmetros e cujos trechos de código são executados no construtor do record. Um detalhe importante é que não podemos usar `this`.

Perceba também que mantivemos o método `isUltimoCapitulo`, porém removendo o prefixo `is`, que é uma convenção de Java Beans, não de records.

Uma coisa ruim é que, ao usarmos records nas classes do domain model do Cotuba, acabamos quebrando a sua API pois não há o prefixo `get` nos métodos de acesso dos atributos. Poderíamos definir manualmente os getters nos records. Porém, se quisermos deixar os records enxutos, teríamos que modificar vários trechos do código.

Para o record `Ebook`, teríamos que:

- modificar `ebook.getArquivoSaida()` para `ebook.arquivoDeSaida()` nas classes `GeradorPDF`, `GeradorEPUB` e `GeradorHTML` do projeto `cotuba-cli`.
- modificar `ebook.getCapitulos()` para `ebook.capitulos()` nas classes `GeradorPDF`, `GeradorEPUB` e `GeradorHTML` do projeto `cotuba-cli` e avisar ao time da Cognitio para que a classe `CalculadoraDeEstatisticas` do projeto `estatisticas-ebook` seja modificada.
- modificar `ebook.isUltimoCapitulo()` para `ebook.ultimoCapitulo()` na classe `GeradorPDF` do projeto `cotuba-cli`.

Já para o record `Capitulo`, teríamos que:

- modificar `capitulo.getTitulo()` para `capitulo.titulo()` nas classes `GeradorEPUB` e `GeradorHTML`.
- modificar `capitulo.getConteudoHTML()` para `capitulo.conteudoHTML()` nas classes `GeradorPDF`, `GeradorEPUB` e `GeradorHTML` do projeto `cotuba-cli` e avisar ao time da Cognitio que devem mudar também a classe `CalculadoraDeEstatisticas` do projeto `estatisticas-ebook`.

O fato de essa pequena mudança requerer alterações em diversos pontos de diferentes projetos mostra o efeito cascata da mudança da API de classes importantes.

Um registro histórico dos records

Um primeiro preview dos records, definido na JEP 359: Records (Preview)(https://openjdk.java.net/jeps/359), foi incluído no Java 14, de março de 2020. No Java 15, lançado em setembro de 2020, houve um segundo preview, definido na JEP 384: Records (Second Preview) (https://openjdk.java.net/jeps/384). Finalmente, os records foram incluídos como funcionalidade da linguagem Java na versão 16, de março de 2021, tendo sido definidos na JEP 395: Records (https://openjdk.java.net/jeps/395).

9.6 Encapsulamento

Digamos que temos uma classe `Tributacao`, conforme a seguir:

```
public class Tributacao {

  public BigDecimal calcula (NotaFiscal nota) {

    //...

    if (nota.getEndereco().getEstado() == Estado.SP &&
        nota.getEmpresa().getSegmento().getSetor()
            == Setor.SERVICOS &&
        nota.getEmpresa().getRegime()
            == RegimeFiscal.LUCRO_PRESUMIDO) {

      BigDecimal uniao = new BigDecimal("3.65");
      BigDecimal imcsEstadual = new BigDecimal("0.0");
      BigDecimal issMunicipal = new BigDecimal("5.0");

      BigDecimal tributacao = uniao.add(imcsEstadual)
          .add(issMunicipal);

      BigDecimal total = BigDecimal.ZERO;
      List<Item> itens = nota.getItens();
      for (Item item : itens) {
        total = total.add(item.getValor());
      }

      BigDecimal tributo = total.multiply(tributacao)
          .divide(new BigDecimal("100"));

      return tributo;

    }

    //...
```

```
  }

}
```

O método `calcula` de `Tributacao` retorna o valor dos tributos para uma dada uma nota fiscal. Para isso, baseia-se:

- no estado;
- no setor e regime fiscal da empresa;
- no valor total dos itens da nota;
- em valores tributários pré-determinados para os âmbitos federal, estadual e municipal.

Qual é o problema com o código anterior? Um dos maus cheiros de código descritos por Kent Beck e Martin Fowler no livro *Refactoring* (FOWLER et al., 1999) é a **Inveja de funcionalidades**:

- *A essência dos objetos é que eles são uma técnica para empacotar dados com os processamentos desses dados.*
- *Um indício clássico de problema é um método que parece mais interessado em uma classe diferente daquela na qual ele se encontra.*
- *O foco mais comum da inveja são os dados.*

A classe `Tributacao` é invejosa. Veja dados de outras classes que são manipulados:

- `nota.getEndereco().getEstado()`
- `nota.getEmpresa().getSegmento().getSetor()`
- `nota.getEmpresa().getRegime()`
- `nota.getItens()`
- `item.getValor()`

O mais problemático, em termos de design de código, é que cálculos e procedimentos sobre esses dados estão sendo feitos fora dos objetos que os contém.

Código invejoso x Código tímido

Para ilustrar o problema de classes invejosas, no artigo *The Art of Enbugging* (HUNT; THOMAS, 2003), Andy Hunt e Dave Thomas citam uma analogia criada por David Bock: *"Suponha que o entregador de jornais vá até a sua porta, demandando o pagamento da semana. Você vira, o entregador puxa a carteira do bolso traseiro da sua calça, tira duas notas e devolve a carteira."*

Um entregador de jornais puxar sua carteira e tirar o que quiser dela não parece uma boa ideia, não é mesmo? O correto seria você mesmo manipular a sua própria carteira, entregando somente as notas que você precisa. A classe `Tributacao` é como o entregador de jornais: está "abrindo a carteira" das classes `Nota` e `Item` e fazendo processamentos que deveriam ser feitos pelas próprias classes.

No mesmo artigo, Andy Hunt e Dave Thomas sugerem que código bom é **tímido**: *"O objetivo fundamental (...) é escrever código tímido: código que não revela muito de si para ninguém e não conversa com os outros mais do que o necessário. Código tímido evita contato com os outros, não é como aquele vizinho fofoqueiro que está envolvido nas idas e vindas de todo mundo. Código tímido nunca mostraria suas coisas “privadas” para os “amigos” (...). Assim como no mundo real, boas cercas fazem bons vizinhos – contanto que você não olhe pela cerca."*

As analogias de código invejoso, do entregador de jornais enxerido e de código tímido trazem uma maneira bem-humorada e ilustrativa de falar de uma ideia muito importante no bom design de código: o **encapsulamento**. Encapsular é esconder, o máximo possível, as informações de uma classe. Assim, evitamos que detalhes de implementação "vazem" para outras classes do sistema.

Encapsulamento x Java Beans

Comumente, estudamos encapsulamento apenas como o uso do modificador *private* nos atributos de uma classe. Estaríamos restringindo a manipulação dos atributos à própria classe. Assim,

não precisaríamos olhar toda a base de código para saber quais trechos manipulam esses atributos.

Atributos privados ajudam, mas não garantem o encapsulamento. Ao definirmos getters e setters, podemos estar compartilhando detalhes da classe indevidamente, facilitando cálculos e processamentos sobre os dados fora da própria classe. Os getters e setters podem levar à inveja no nosso design!

O triste é constatar que várias bibliotecas da plataforma Java requerem Java Beans, que são classes com atributos privados, getters e setters e o construtor sem parâmetros. As bibliotecas usam esse formato comum para manipular, via Reflection API, os Java Beans. Mas o design acaba direcionado a um mau caminho.

Mesmo evitando setters, ainda não há garantia de encapsulamento: perceba que a classe `Tributacao` usa apenas getters. Porém, obtém detalhes das outras classes e realiza lógica de negócio com os dados obtidos.

Encapsulamento x Herança

Conforme descrito no livro *Design Patterns* (GAMMA et al., 1994), há um conflito entre herança e encapsulamento: *"Como a herança expõe uma subclasse a detalhes da implementação de sua superclasse, costuma-se dizer que "a herança quebra o encapsulamento. A implementação de uma subclasse fica tão ligada à implementação de sua superclasse que qualquer alteração na implementação da superclasse forçará a subclasse a mudar."*

Arquitetura Negativa

Imutabilidade e encapsulamento são exemplos de restrições que impomos ao nosso código que o tornam melhor de entender e manter. Com imutabilidade minimizamos os pontos em que um objeto pode ter seu estado modificado e com encapsulamento minimizamos a "superfície" compartilhada entre os objetos.

Michael Feathers, em seu artigo *Negative Architecture* (FEATHERS, 2018), chama essas restrições de *Arquitetura Negativa*: *"Você pode olhar para essa área do seu código e saber que tem uma coisa a menos para pensar/preocupar-se. (...) Saber o que algo NÃO é capaz de fazer reduz o número de armadilhas. Isso coloca a arquitetura em terra firme."*

A Lei de Deméter e o lema "Diga, não pergunte"

Se atributos privados e a ausência de setters não garantem o encapsulamento, o que fazer? Andy Hunt e Dave Thomas, em seu livro *Pragmatic Programmer* (HUNT; THOMAS, 1999) citam o trabalho de Ian Holland na Northeastern University que, por volta de 1987, cunhou a **Lei de Deméter**. Essa "lei" afirma que todo método de um objeto deve chamar apenas métodos pertencentes a:

- si mesmo
- quaisquer parâmetros que foram passados para o método
- quaisquer objetos criados
- qualquer composição

É uma maneira mais detalhada de declarar que um bom objeto apenas interage com seus "vizinhos" imediatos.

> Curiosidade: Deméter ou Demetra é a deusa grega da agricultura. Era o nome do projeto da Northeastern University que deu origem ao termo.

A classe `Tributacao` infringe a lei de Deméter porque chama métodos de objetos que não são seus colaboradores imediatos. Um exemplo é o trecho `nota.getEmpresa().getSegmento().getSetor()`. Obtemos o setor do segmento da empresa da nota fiscal. É muita inveja e intromissão na vida alheia!

O fato de percorrermos os itens da nota fiscal para calcular o valor total é outra violação dessa lei. Um item é um detalhe da nota. A

classe `Tributacao` não deveria lidar com itens e nem sequer saber que eles existem!

No artigo *The Art of Enbugging* (HUNT; THOMAS, 2003), Andy Hunt e Dave Thomas indicam que um lema de OO deveria ser **Tell, don't Ask**. Em português, algo como "Diga, não pergunte": *"Envie comandos para objetos dizendo o que você quer fazer. Explicitamente, não queremos consultar um objeto sobre seu estado, tomar uma decisão e, então, dizer ao objeto o que fazer."*

> *Perceber se um código está bem encapsulado, ou não, não é tão difícil.*
>
> *(...) se pergunte:*
>
> *O que esse método faz? Provavelmente sua resposta será: eu sei o que o método faz pelo nome dele (...)*
>
> *Como ele faz isso? Sua resposta provavelmente é: se eu olhar só para esse código, não dá para responder."*
>
> Maurício Aniche, no livro *OO e SOLID para Ninjas* (ANICHE, 2015)

9.7 Design Pattern: Iterator

O time da Cognitio, com a ideia de encapsulamento em mente, reparou no código da classe `CalculadoraDeEstatisticas` do projeto `estatisticas-ebook`:

```
// br.com.cognitio.estatisticas.CalculadoraDeEstatisticas

public class CalculadoraDeEstatisticas
    implements AoFinalizarGeracao {

@Override
public void aposGeracao(Ebook ebook) {

  var contagemDePalavras = new ContagemDePalavras();

  // código omitido...

  for (Map.Entry<String, Integer> contagem :
      contagemDePalavras.entrySet()) {
    String palavra = contagem.getKey();
    Integer ocorrencias = contagem.getValue();
    System.out.println(palavra + ": " + ocorrencias);
  }

}
```

São realizados os seguintes passos:

- um objeto da classe `ContagemDePalavras` é instanciado;
- o objeto é preenchido com as palavras extraídas do ebook, omitido no trecho anterior;
- é obtido um `Set` com os pares chave-valor através do método `entrySet`;
- o `Set` é percorrido em um *for-each*.

O método `entrySet` retorna o `Set` de `Map.Entry` do mapa usado internamente por `ContagemDePalavras`:

```
// br.com.cognitio.estatisticas.ContagemDePalavras

class ContagemDePalavras {

  private Map<String, Integer> map = new TreeMap<>();

  // código omitido...

  Set<Map.Entry<String, Integer>> entrySet() {
    return map.entrySet();
  }

}
```

A conclusão do time da Cognitio foi que o encapsulamento de `ContagemDePalavras` é quebrado no método `entrySet`. O `Map.Entry` é um detalhe de implementação que é vazado para as classes que usam esse método. Para resolver esse problema, seria interessante ser possível percorrer com um *for-each* o próprio objeto da classe `ContagemDePalavras`.

Design Pattern: Iterator

O acesso às estruturas de dados usadas internamente, seja `List`, `Set` ou qualquer outra, deve ficar encapsulado ao objeto que as define. Não deve haver vazamentos. Mas como acessar o conteúdo dessas estruturas de dados sem expô-las?

O livro *Design Patterns* (GAMMA et al., 1994) cataloga o pattern **Iterator**, uma maneira de acessar sequencialmente um objeto agregado, como uma lista, sem expor sua representação interna.

Na API de Collections do Java, há a interface `java.util.Iterator`, que define os métodos:

- `hasNext` , que retorna um `boolean` indicando se há mais elementos
- `next` , que retorna o próximo elemento da iteração

Várias das Collections do Java, como `List` e `Set` , possuem métodos que retornam um `Iterator` . Isso também vale para o `entrySet` de um `Map` .

O time da Cognitio pode remover o método `entrySet` de `ContagemDePalavras` e definir um método `iterator` que retorna o `Iterator` obtido a partir do mapa que é usado internamente pela classe:

```
// br.com.cognitio.estatisticas.ContagemDePalavras

import java.util.Iterator; // inserido

class ContagemDePalavras {

  // código omitido...

   Set<Map.Entry<String, Integer>> entrySet() {
    return map.entrySet();
  }

  // inserido
  public Iterator<Map.Entry<String, Integer>> iterator() {
    return this.map.entrySet().iterator();
  }

}
```

Porém, o `Iterator` ainda é definido em teremos de `Map.Entry` e, portanto, ainda está relacionado a um detalhe interno de `ContagemDePalavras` . No lugar do `Map.Entry` , seria interessante definir uma classe de negócio que contenha uma palavra com a quantidade de ocorrências associada. É possível defini-la como um record aninhado dentro de `ContagemDePalavras` :

```
// br.com.cognitio.estatisticas.ContagemDePalavras

class ContagemDePalavras {

  static record Contagem(String palavra, int ocorrencias) {
  }

  // código omitido...

}
```

Então, o `Iterator<Map.Entry<String, Integer>>` poderia ser trocado por um `Iterator<Contagem>`. Para isso, seria necessário fornecer uma implementação da interface `Iterator`, que pode ser uma classe anônima:

```
// br.com.cognitio.estatisticas.ContagemDePalavras

class ContagemDePalavras {

  // código omitido...

  public Iterator<Contagem> iterator() {

    Iterator<Map.Entry<String, Integer>> iterator =
        this.map.entrySet().iterator();

    return new Iterator<>() {

      @Override
      public boolean hasNext() {
        return iterator.hasNext();
      }

      @Override
      public Contagem next() {
        Map.Entry<String, Integer> entry = iterator.next();
        String palavra = entry.getKey();
        int ocorrencias = entry.getValue();
        return new Contagem(palavra, ocorrencias);
      }
```

```
    };
  }

}
```

> Observação: não é possível definir um lambda porque `Iterator` não é uma interface funcional, já que é necessário definir mais de um método.

O `Iterator<Map.Entry<String, Integer>>` ficaria escondido internamente na classe `ContagemDePalavras`, na implementação dos métodos `hasNext` e `next`. No método `next`, um objeto da classe `Contagem` é instanciado a partir do par chave-valor obtido.

Iteração externa x iteração interna

Na classe `CalculadoraDeEstatisticas`, que usa `ContagemDePalavras`, o `Iterator` seria percorrido usando os métodos `hasNext` e `next`:

```
// br.com.cognitio.estatisticas.CalculadoraDeEstatisticas

Iterator<ContagemDePalavras.Contagem> iterator =
    contagemDePalavras.iterator();

while (iterator.hasNext()) {
  ContagemDePalavras.Contagem contagem = iterator.next();

  System.out.println(contagem.palavra() + ": " +
      contagem.ocorrencias());
}
```

Quando o cliente, ou seja, a classe que usa o Iterator controla a obtenção dos valores temos uma **iteração externa**. A alternativa é a **iteração interna**, em que o próprio Iterator controla os valores que serão fornecidos para o cliente.

A plataforma Java, desde a versão 5, possui a interface `java.lang.Iterable`. Classes que implementam essa interface podem

ser usadas diretamente em um *for-each*. Apenas um método é definido: o `iterator`, que retorna um `Iterator`.

Seria possível definir a classe `ContagemDePalavras` como implementação de `Iterable<ContagemDePalavras.Contagem>`:

```
// br.com.cognitio.estatisticas.ContagemDePalavras

class ContagemDePalavras
    implements Iterable<ContagemDePalavras.Contagem> {
  // código omitido...
}
```

Como já havia um método `iterator` definido anteriormente, a interface já estaria implementada. Na classe `CalculadoraDeEstatisticas`, que usa `ContagemDePalavras`, a iteração poderia ser simplificada, deixando apenas o *for-each*:

```
// br.com.cognitio.estatisticas.CalculadoraDeEstatisticas

for (ContagemDePalavras.Contagem contagem : contagemDePalavras) {
  System.out.println(contagem.palavra() + ": " +
      contagem.ocorrencias());
}
```

Dessa maneira, não há nenhuma menção a detalhes internos da classe `ContagemDePalavras`. Apenas são adicionadas palavras e os resultados são percorridos, obtendo cada contagem. Não dá pra saber como isso é feito, apenas o que é feito. Detalhes encapsulados!

9.8 O que aprendemos?

Neste capítulo, vimos como a imutabilidade pode ser usada para limitar os pontos que alteram o estado de nossos objetos e o encapsulamento pode ser usado para minimizar o compartilhamento de detalhes internos entre os objetos. Estudamos também design patterns como o Builder e Iterator, além de records.

No próximo capítulo, veremos o que são módulos e quais os princípios de Uncle Bob para termos alta coesão e baixo acoplamento em nossos módulos!

O código deste capítulo pode ser encontrado em: https://github.com/alexandreaquiles/solid-na-pratica/tree/cap9-imutabilidade-e-encapsulamento

Figura 9.1: Vídeo do capítulo

CAPÍTULO 10

Princípios de Coesão e Acoplamento de Módulos

10.1 O que são módulos?

À medida que aplicações crescem em tamanho e complexidade, é necessária alguma maneira de organizar o código para além de classes. Uma aplicação pode ser separada em módulos, que são "fatias" independentes de código.

Kirk Knoernschild, no livro *Java Application Architecture: Modularity Patterns* (KNOERNSCHILD, 2012), explora algumas características dos módulos. Módulos são:

- **Implantáveis**: são entregáveis que podem ser executados em *runtime*.
- **Reusáveis**: são nativamente reusáveis por diferentes aplicações, sem a necessidade de comunicação pela rede. As funcionalidades de um módulo são invocadas diretamente, dentro da mesma JVM e, portanto, do mesmo processo (no Windows, o mesmo `java.exe`).
- **Testáveis**: podem ser testados independentemente, com testes de unidade.
- **Gerenciáveis**: em um sistema de módulos, podem ser instalados, reinstalados e desinstalados.
- **Componíveis**: podem se unir a outros módulos para compor uma aplicação.
- **Sem estado**: classes podem ter estado, módulos não.

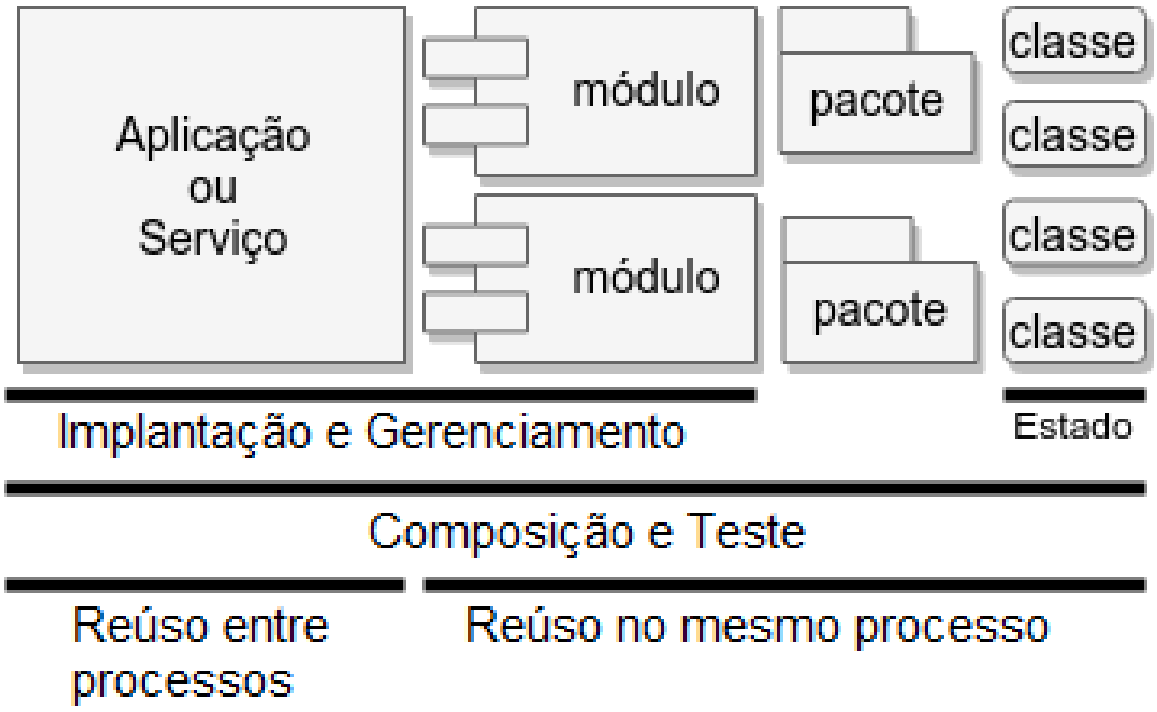

Figura 10.1: Características de módulos

Qual será o artefato Java que contém todas essas características?

JARs

Em Java, código é definido em classes. Classes ficam organizadas dentro de pacotes. E qual é a maneira de agrupar pacotes? É o JAR, ou **J**ava **AR**chive.

JARs são arquivos compactados no formato ZIP que contêm pacotes que, por sua vez, contêm os `.class` compilados a partir do código-fonte das classes. Um JAR é implantável, reusável, testável, gerenciável, sem estado e é possível compô-lo com outros JARs para formar uma aplicação.

Kirk Knoernschild define: *O melhor candidato à unidade de modularidade na plataforma Java é o arquivo JAR!* (KNOERNSCHILD, 2012)

Portanto, no Java, **um módulo é um JAR**.

Por que modularizar?

No livro *Modular Java* (WALLS, 2009), Craig Walls cita algumas vantagens de modularizar uma aplicação:

- capacidade de trocar um módulo por outro, com uma implementação diferente, desde que a interface pública seja mantida;
- facilidade de compreensão de cada módulo individualmente;
- possibilidade de desenvolvimento em paralelo, permitindo que tarefas sejam divididas entre diferentes times;
- testabilidade melhorada, permitindo um outro nível de testes, que trata um módulo como uma unidade;
- flexibilidade, permitindo o reúso de módulos em outras aplicações.

O que é reúso?

No artigo *Granularity* (MARTIN, 1996e), Uncle Bob diz que **copiar e colar código dos outros não é reúso** porque você se torna dono do código que você copia:

- se algo tiver que ser adaptado, você deve alterar o código;
- se você achar bugs, você deve corrigi-los;
- se o autor original encontrar bugs, você que deverá ficar sabendo e descobrir o que deve ser alterado na sua cópia.

Copiar e colar pode ser mais fácil em um primeiro momento, mas é muito caro no longo prazo, em que manutenções são frequentes.

No mesmo artigo, Uncle Bob diz: *"Eu reúso código se, e somente se, eu nunca preciso olhar para o código-fonte. (...) Ou seja, eu espero que o código que eu estou reusando seja tratado como um produto. Não é mantido por mim. Não é distribuído por mim. Eu sou o consumidor. O autor, ou outra entidade, é responsável por mantê-lo."*

Reúso não é algo específico de linguagens OO, mas mecanismos como interfaces e polimorfismo facilitam a criação de APIs reutilizáveis de maneira consistente. Reutilizar código de terceiros, como bibliotecas e frameworks, permite um ganho enorme de produtividade. Imagine a complicação de um mundo sem JPA/Hibernate ou Spring, por exemplo.

Também é possível reaproveitar código entre diferentes aplicações de uma mesma empresa/organização. Quanto maior a base de código, maior é a chance de reúso.

A cópia do Micronaut

Um caso interessante é o código do módulo de controle transacional do Micronaut, o `micronaut-data-tx`. Esse módulo é um fork do módulo `spring-tx` do Spring com modernizações como o uso de enums e de logs com SLF4J, além de adaptações para trocar dependências ao Spring pelo Micronaut.

Por exemplo, as classes `AbstractPlatformTransactionManager` do `spring-tx` e `AbstractSynchronousTransactionManager` do `micronaut-data-tx` são bastante semelhantes. Os desenvolvedores do Micronaut, entre eles Graeme Rocher, aproveitaram bastante código complexo mas, ao copiar o código, passaram a ter a responsabilidade de mantê-lo.

10.2 Princípios de coesão de módulos

Ao pensarmos em módulos, uma importante questão é: como "fatiar" o código para definir o que deve ficar em cada módulo? Ou melhor: quais classes devem ficar em quais módulos? Quais as **responsabilidades** de um módulo?

Coesão, algo importante no nível de classes, também é importante no nível de módulos. Por isso, Uncle Bob, define alguns princípios relativos à **coesão de módulos**.

O Princípio da Equivalência entre Entrega e Reúso (REP)

O reúso de um módulo pode ser feito por terceiros ou por outros times da mesma empresa. Mas como um módulo pode ser entregue para reúso? Pode ser através de:

- um site próprio;
- pelo GitHub, GitLab ou equivalente;
- por uma ferramenta como Maven ou Gradle, no repositório central ou em um gerenciador de artefatos como o Sonatype Nexus ou JFrog Artifactory.

O código do Cotuba foi disponibilizado para as empresas Paradizo e a Cognitio, que implementaram plugins, diretamente pelo GitHub. As equipes de ambas as empresas têm que clonar o repositório do `cotuba-cli` em sua última versão, compilá-lo e instalá-lo manualmente no repositório local do Maven. Um fluxo mais interessante para as parceiras seria a entrega do `cotuba-cli` no repositório central do Maven ou em algum gerenciador de artefatos.

Uma vez fechado um grupo de alterações no código, acontece o processo de compilação, teste, documentação e entrega dos módulos, publicando uma nova versão. Quem usa esse módulo, deve ser avisado sobre a nova versão.

Só os módulos que são versionados, entregues e disponibilizados por quem os mantém podem ser considerados reusáveis.

Além disso, não disponibilizamos classes diretamente. O que usamos são os módulos, que são agrupamentos de classes. Usamos apenas a totalidade de um módulo. Não é possível usar apenas parte do que está contido em um módulo. Um módulo é atômico. Indivisível. Essa constatação é descrita por Uncle Bob no seguinte princípio:

> **Release/Reuse Equivalency Principle (REP)**
>
> *A granularidade de reúso é a granularidade de entrega.*

O Princípio do Agrupamento Comum (CCP)

Mudanças em uma aplicação são necessárias. Se o mundo muda, a aplicação muda. Para isolar o impacto dessas mudanças no nível de classes, temos o SRP. O SRP nos diz que devemos agrupar, em uma classe, apenas código que tem o mesmo motivo para ser modificado. Mas e no nível de módulos?

Quando há necessidade de uma modificação na aplicação, o ideal é que uma alteração seja isolada no código de apenas um módulo. Dessa forma, é minimizado o trabalho de publicar uma nova versão: ao invés de entregar vários módulos, será entregue apenas um.

Uncle Bob definiu um princípio que é o equivalente ao SRP para módulos:

COMMON CLOSURE PRINCIPLE (CCP)

Agrupe em módulos as classes que são modificadas pelos mesmos motivos e ao mesmo tempo. Separe em módulos diferentes as classes que são modificadas em momentos e por motivos diferentes.

O ideal é que uma mudança ocasione a alteração do código de apenas um módulo. Mas, na prática, pode acontecer de termos que modificar código de vários módulos.

O agrupamento das classes deve ser feito considerando as mudanças mais prováveis de acontecer. Desejamos minimizar o número de módulos a serem modificados. A maneira de atingir isso vai depender do problema que estamos resolvendo. E, claro, podemos errar.

Há uma forte sinergia entre o CCP e o critério de David Parnas no artigo *On the Criteria To Be Used in Decomposing Systems into Modules* (PARNAS, 1972). Parnas indica que módulos que seguem o critério de *information hiding*, ou seja, que minimizam o compartilhamento de decisões internas, teriam menos impacto de mudanças no projeto.

O Princípio do Reúso Comum (CRP)

O ISP define que classes não devem depender de métodos que não usam. Isso implica em interfaces coesas, que definem contratos menores e, também, em usar interfaces para expor o mínimo possível de uma classe. Qual o equivalente no nível de módulos?

Não deveríamos colocar, em um mesmo módulo, classes que seriam utilizadas apenas por parte dos clientes desse módulo. Uma mudança em uma classe forçaria a publicação de uma nova versão do módulo já que não é possível entregar apenas parte de um módulo.

Uma nova versão requereria recompilação, reteste e reimplantação de todos que usam o módulo. Mesmo daqueles que não usam a classe modificada.

O equivalente ao ISP para módulos foi definido por Uncle Bob no seguinte princípio:

> **COMMON REUSE PRINCIPLE (CRP)**
>
> *As classes de um módulo são reusadas em grupo. Quem reúsa uma dessas classes reúsa todas.*
>
> ou
>
> *Não force quem usa um módulo a depender de coisas de que eles não precisam.*

A tensão entre os princípios de coesão de módulos

No livro *Clean Architecture*, Uncle Bob diz que há uma tensão entre o REP, o CCP e o CRP (MARTIN, 2017):

- REP agrupa classes para facilitar o reúso por outros projetos.
- CCP agrupa classes para minimizar impactos de mudanças.
- CRP divide módulos em menores, pensando em evitar dependências e entregas desnecessárias.

O REP e CCP levariam à inclusão de mais classes, tendendo a criar módulos maiores, com granularidade mais grossa. Porém, os motivos de agrupamento são diferentes: o REP por reúso, o CCP, por mudanças. Já o CRP levaria à exclusão de classes, tendendo a criar módulos menores, com granularidade mais fina.

Se a granularidade for muito grossa, com muitas classes em poucos módulos, uma mudança levará a novas versões sem muitas novidades para boa parte de quem usa esses módulos. Se, por outro lado, a granularidade dos módulos for muito fina, com poucas

classes em muitos módulos, o impacto de uma mudança levará a alterações em diversos módulos diferentes.

Saber qual princípio deve ser favorecido é uma decisão que deve levar em conta o contexto do projeto. E à medida que o projeto avança, o contexto muda e as decisões devem ser reavaliadas.

Módulos, componentes ou pacotes?

Em seus textos mais antigos, Uncle Bob usava o termo *packages*, ou *pacotes*, para se referir ao que chamamos de módulos.

Porém, pacote é um termo com um significado bem definido na plataforma Java, que define tanto a estrutura de diretórios como o nome completo (ou *fully qualified name*) de uma classe. Em outras tecnologias OO, como C++ e C#, esse tipo de agrupamento de classes é conhecido como *namespace*.

No artigo *The Principles of OOD* (MARTIN, 2005), esclarece a confusão: *"(No contexto dos princípios de pacotes,) um pacote é um entregável binário como um arquivo* `.jar` *ou um* `dll`*, em oposição a um* `package` *Java ou a um* `namespace` *C++."* Em livros mais recentes, Uncle Bob usa o termo *binary components* ou simplesmente *components* ou, em português, *componentes*.

A partir do JDK 9, há uma tecnologia de componentização, o JPMS, que usa o termo **módulos**. É o termo que estamos utilizando. Estudaremos essa nova tecnologia em capítulos posteriores.

10.3 Modularizando o Cotuba

Os módulos da plataforma Java são JARs. Já abordamos alguns módulos:

- `cotuba-cli` : faz a renderização de MD para HTML, a geração ebooks nos formatos de EPUB, PDF e HTML, tem uma interface de linha de comando, expõe um *hook* na renderização, outro *hook* na finalização de ebook, além da lógica para integrar isso tudo.
- `tema-paradizo` : o plugin de temas da empresa Paradizo.
- `estatisticas-ebook` : o plugin que gera estatísticas da empresa Cognitio.

As implementações dos plugins `AoRenderizarHTML` no módulo `tema-paradizo` e `AoFinalizarGeracao` no módulo `estatisticas-ebook` são bem coesas, com responsabilidades bem definidas.

Porém, será que o módulo `cotuba-cli` respeita o CCP? Será que contém apenas classes que tem os mesmos motivos para serem modificadas? Uma alteração no código do módulo `cotuba-cli` pode ser necessária por:

- mudanças na interface de linha de comando e/ou na biblioteca Apache Commons CLI
- mudanças na renderização do MD e/ou na biblioteca CommonMark Java
- mudanças na geração do PDF e/ou na biblioteca iText
- mudanças na geração do EPUB e/ou na biblioteca Epublib
- mudanças na geração de HTML

São muitos motivos para mudar o `cotuba-cli` . Precisamos fatiá-lo! E há muitas maneiras de fatiar esse módulo, separando classes que mudam por diferentes razões.

Um módulo por pacote

Uma ideia seria ter um módulo para cada pacote. Isso levaria a vários módulos com responsabilidades bem pequenas:

- `cotuba-cli` , responsável pela interface da linha de comando
- `cotuba-application` , contendo a classe `Cotuba` e demais abstrações de alto nível
- `cotuba-domain` , contendo código de domínio
- `cotuba-md` , que renderiza os arquivos MD para HTML
- `cotuba-pdf` , que gera ebooks no formato PDF
- `cotuba-epub` , que gera ebooks no formato EPUB
- `cotuba-html` , que gera ebooks no formato HTML
- `cotuba-plugin` , que expõe os plugins

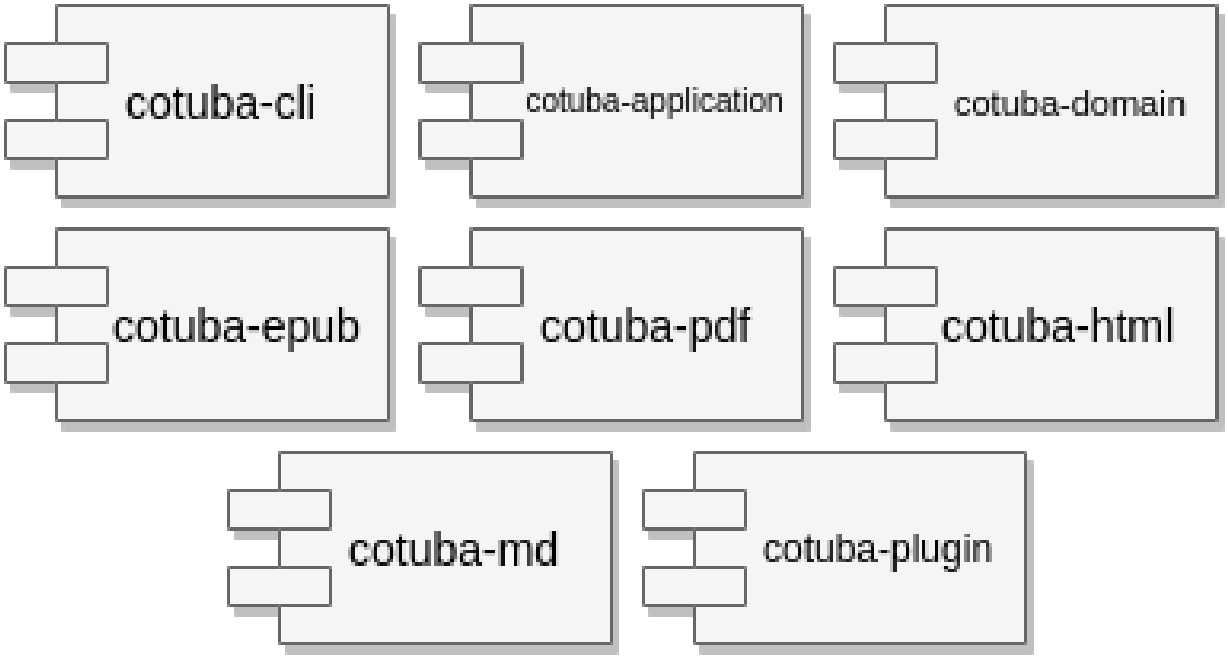

Figura 10.2: Um módulo para cada pacote

Módulos com responsabilidades bem definidas respeitam o CCP: teríamos agrupado classes que mudam pelo mesmo motivo. Além disso, como "pulverizamos" o código em vários módulos minúsculos, uma aplicação poderia utilizar apenas partes dos módulos. Por

exemplo, para renderizar arquivos MD, usaria (ou melhor, reusaria) o módulo `cotuba-md`.

O módulo de renderização de MDs, por sua vez, usaria os módulos `cotuba-application` e `cotuba-domain`. E o módulo `cotuba-domain` acabaria usando classes dos módulos `cotuba-pdf`, `cotuba-epub` e `cotuba-html` em sua enum `FormatoEbook`, por incrível que pareça:

```
// cotuba.domain.FormatoEbook

import cotuba.epub.GeradorEPUB; // seria do módulo `cotuba-epub`
import cotuba.pdf.GeradorPDF; // seria do módulo `cotuba-pdf`
import cotuba.html.GeradorHTML; // seria do módulo `cotuba-html`

public enum FormatoEbook {

// restante do código...
```

O reúso de `cotuba-md` acabaria forçando o reúso de quase todos os módulos por causa de suas **dependências transitivas**, as dependências de suas dependências.

Será que essa solução atende ao CRP? Não! Para atender, não deveríamos ter classes não utilizadas ao reusar um módulo. Porém, ao reusar o módulo `cotuba-md`, por suas dependências transitivas, teríamos também classes para gerar ebooks em PDF, EPUB e HTML. Inúteis para quem quer apenas renderizar Markdown!

E quanto à granularidade da entrega referenciada pelo REP? Muito fina: seriam 8 módulos! Teríamos que controlar versões, fazer o build e entregar todos esses módulos, o que é bastante trabalhoso de manter. E, extrapolando um pouco o REP, se a entrega for difícil, o reúso é difícil.

Resumindo, em relação a essa ideia de um módulo por pacote teríamos:

- REP: entrega e reúso dificultados pelo grande número de módulos.

- CCP: atendido com louvor, pelos módulos focados e com responsabilidades bem definidas.
- CRP: não atendido porque acabamos dependendo transitivamente de módulos cujo código é desnecessário.

Um módulo separado para a UI

Uma outra ideia seria ter um módulo separado para a UI e outro para o resto da aplicação. Teríamos módulos como os a seguir:

- `cotuba-cli` , responsável pela interface da linha de comando
- `cotuba-core` , responsável por todo o resto, como a renderização do MD para HTML e a geração ebooks nos formatos PDF, EPUB e HTML

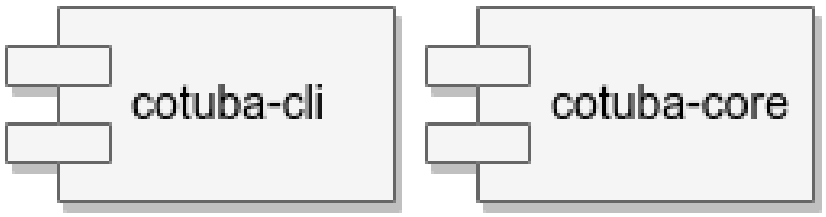

Figura 10.3: Um módulo para cada pacote

O módulo `cotuba-cli` teria uma responsabilidade bem delimitada. Já o `cotuba-core` teria toda a lógica de geração dos ebooks. Em termos de CCP, aumentaríamos os motivos para mudança desse último módulo.

Se uma aplicação desejar o reúso da renderização de MDs, teria que depender do `cotuba-core` como um todo. E acabaria levando junto código desnecessário. Ou seja, o CRP não é atendido.

Já o REP é favorecido: apenas um módulo, o `cotuba-core` , deve ser versionado e entregue para que possa haver reúso da renderização de MDs. Com a interface de linha de comando, seriam 2 módulos.

Resumindo, com a estratégia de separar a UI em módulo próprio, teríamos:

- REP: bastante facilitado em relação à solução anterior, já que há a entrega, e o respectivo reúso, de apenas 2 módulos.
- CCP: menos favorecido que a solução anterior pois há um módulo bem focado na UI e outro mais geral, com mais razões para ser modificado.
- CRP: não atendido, da mesma maneira que a solução anterior, porque acabamos com código desnecessário no módulo geral.

Um módulo para cada entrada e saída

Há ainda outra ideia: separar, em módulos específicos, o código das entradas, das saídas e das regras de negócio. No caso do Cotuba, quais seriam as entradas e saídas? A entrada é feita por meio de parâmetros na UI de linha de comando. Já as saídas são os ebooks gerados. Por isso, teríamos os módulos:

- `cotuba-cli` , responsável pela interface da linha de comando
- `cotuba-pdf` , com código para a geração de PDF
- `cotuba-epub` , com código para a geração de EPUB
- `cotuba-html` , com código para a geração de HTML
- `cotuba-core` , que contém a renderização do MD, invocação dos plugins e lógicas de negócio

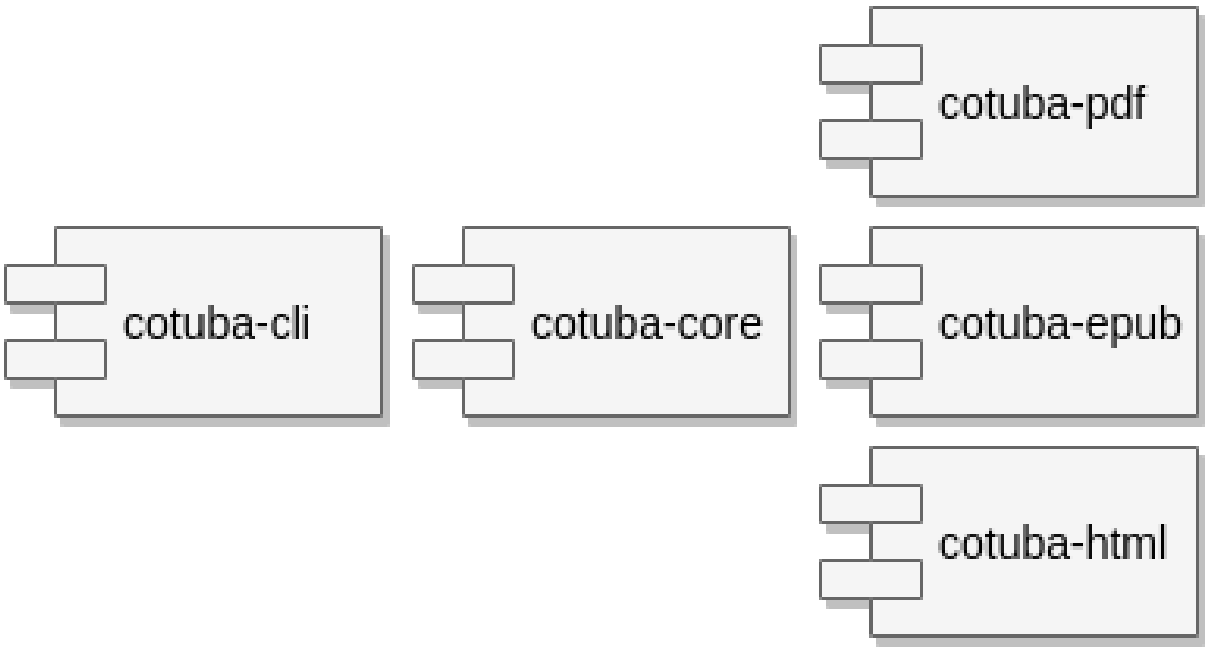

Figura 10.4: Módulos separados para a UI e para os geradores de ebook

Em termos de CCP, há uma distribuição de responsabilidades bem razoável, com módulos focados em cada possível mudança. Se, por exemplo, a taxa de modificações na linha de comando for frequente, mudaríamos apenas o módulo `cotuba-cli`. Alterações no PDF, afetariam somente o módulo `cotuba-pdf`.

No caso de reúso da renderização de MDs, dependeríamos do módulo `cotuba-core`, com tudo o que há dentro dele. Ainda precisaríamos dos módulos que geram os ebooks em diferentes formatos, já que são dependências transitivas da enum `FormatoEbook` que estaria em `cotuba-core`.

Caso o reúso pretendido for da geração de PDFs, por exemplo, precisaríamos do módulo `cotuba-pdf`. Também precisaríamos do `cotuba-core`, cujas classes e interfaces são usadas pelo gerador de PDFs. Como `cotuba-core` pode disparar a geração de ebooks nos formatos EPUB e HTML, precisaríamos também do módulo `cotuba-epub` e `cotuba-html`.

Portanto, quanto ao REP, o versionamento e entrega de apenas um módulo, o `cotuba-core`, permitiria o reúso da renderização de MDs.

Ao todo, teríamos 5 módulos.

Em relação aos princípios de coesão de módulos, teríamos:

- REP: um meio termo, com entrega, e possível reúso, de 5 módulos.
- CCP: também um meio termo, com módulos mais focados do que ao separarmos apenas a UI, mas menos focados do que um módulo por pacote.
- CRP: não atendido, porque ao tentar reusar o módulo geral ou cada módulo dos geradores, acabaríamos precisando de quase todos os módulos.

Escolhendo uma estratégia de modularização

Qual estratégia escolher? Podemos:

- deixar tudo num módulo só, como está;
- ter módulos por pacote, "pulverizando" o código em 8 módulos;
- manter um módulo para a UI de linha de comando, outro para o resto;
- definir um módulo para a entrada (UI de linha de comando), um para cada saída (geradores de PDF, EPUB e HTML) e outro para a coordenação da geração de ebooks.

Para balancear entre reusabilidade de apenas parte dos módulos, coesão, manutenibilidade e simplicidade no desenvolvimento, versionamento e entrega, vamos usar a estratégia de **um módulo separado para UI**. Portanto, teríamos 2 módulos: o `cotuba-cli` e o `cotuba-core`.

A questão é: como definir um módulo em um projeto Java?

Uma das alternativas é definir os módulos como projetos independentes em um *monorepo*, uma estratégia em que todo o código de uma organização fica em um mesmo repositório da ferramenta de controle de versão. Essa estratégia é adotada por

empresas com bases de código massivas como Google, Facebook, Uber e Twitter.

Outra alternativa é utilizar os recursos de modularização do Maven. É a solução que adotaremos!

10.4 Módulos Maven

Com o Maven, é possível criarmos um **multi-module project**, que permite definir vários módulos em um mesmo projeto. O Maven ficaria responsável por obter as dependências necessárias e o fazer *build* na ordem correta. Os artefatos gerados (JARs, WARs e/ou EARs) teriam a mesma versão. Devemos definir um módulo pai, ou supermódulo, que contém um ou mais módulos filhos, ou submódulos.

Por exemplo, poderíamos ter uma aplicação Web para a editora Casa do Código definida com o supermódulo `casadocodigo` e os submódulos `casadocodigo-web` e `casadocodigo-core`. A estrutura de diretórios seria a seguinte:

```
casadocodigo
│
├── pom.xml
│
├── casadocodigo-web
│   ├── pom.xml
│   └── src
│
└── casadocodigo-core
    ├── pom.xml
    └── src
```

O supermódulo `casadocodigo` definiria um `pom.xml`. Nesse arquivo, a propriedade `packaging` teria o valor `pom`. Também seriam definidas propriedades, dependências, repositórios e outras configurações comuns a todos os submódulos. Esses submódulos seriam declarados da seguinte maneira:

```
<modules>
  <module>casadocodigo-web</module>
```

```
  <module>casadocodigo-core</module>
</modules>
```

Já os submódulos não definiriam um `<groupId>` nem um `<version>` próprios, apenas o `<artifactId>`. O supermódulo deveria ser declarado com a tag `<parent>`. Segue o exemplo para o módulo Web da Casa do Código:

```
<parent>
  <groupId>casadocodigo</groupId>
  <artifactId>casadocodigo</artifactId>
  <version>0.0.1-SNAPSHOT</version>
</parent>

<artifactId>casadocodigo-web</artifactId>
```

Os arquivos `pom.xml` dos submódulos definiriam em seus `pom.xml` suas próprias configurações, como propriedades, dependências e repositórios. Se houvesse uma dependência a outros módulos do supermódulo, seria possível usar `${project.version}` como versão. Por exemplo, o submódulo `casadocodigo-web` poderia declarar a dependência ao submódulo `casadocodigo-core` conforme o código que segue:

```
<dependency>
  <groupId>casadocodigo</groupId>
  <artifactId>casadocodigo-core</artifactId>
  <version>${project.version}</version>
</dependency>
```

Agora que sabemos o que devemos fazer, mãos à obra!

Criando um projeto Maven multimódulos para o Cotuba

No Cotuba, podemos criar dois submódulos separados: um para a UI, outro para o resto. Primeiramente, vamos mover o código atual do Cotuba para um diretório `cotuba-cli-backup`:

```
mv cotuba cotuba-cli-backup
```

É importante abrir o projeto novamente em sua IDE.

Em seguida, vamos definir um supermódulo `cotuba`, que deve definir um `pom.xml` com as seguintes propriedades:

- *Group Id*: `cotuba`
- *Artifact Id*: `cotuba`
- *Version* como `0.0.1-SNAPSHOT`.
- *Packaging* como `pom`.
- a codificação de caracteres como UTF-8
- o Java 17 tanto para o código-fonte como para o código compilado

Também é importante configurar o `maven-compiler-plugin` para que seja definida a versão `3.10.0`. O `pom.xml` do supermódulo `cotuba` deve ficar algo semelhante a:

```
<!-- cotuba/pom.xml -->

<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>cotuba</groupId>
  <artifactId>cotuba</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <packaging>pom</packaging>

  <properties>
    <project.build.sourceEncoding>UTF-8
      </project.build.sourceEncoding>
    <maven.compiler.source>17</maven.compiler.source>
    <maven.compiler.target>17</maven.compiler.target>
  </properties>

  <build>
```

```
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-compiler-plugin</artifactId>
        <version>3.10.0</version>
      </plugin>
    </plugins>
  </build>

</project>
```

As configurações no supermódulo serão aplicadas por padrão a todos os submódulos.

Criando o módulo cotuba-core

Vamos criar um módulo chamado `cotuba-core` como um submódulo de `cotuba`. Deverá conter quase todo o código do Cotuba, com exceção do código relacionado à UI de linha de comando.

> Dica: use o auxílio da sua IDE!

Devem ser feitas as seguintes configurações:

- o `<parent>` deve ser definido com `cotuba` tanto no `<groupId>` como no `<artifactId>` e com `0.0.1-SNAPSHOT` na `<version>`
- o `<artifactId>` do módulo deve ser definido como `cotuba-core`

As demais configurações, como versão do Java e codificação de caracteres, serão herdadas do supermódulo `cotuba`. O `pom.xml` do módulo `cotuba-core` ficaria algo como:

```
<!-- cotuba/cotuba-core/pom.xml -->

<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    http://maven.apache.org/xsd/maven-4.0.0.xsd">
```

```
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>cotuba</groupId>
    <artifactId>cotuba</artifactId>
    <version>0.0.1-SNAPSHOT</version>
  </parent>
  <artifactId>cotuba-core</artifactId>

</project>
```

A estrutura de diretórios deve ser a seguinte:

```
cotuba
│
├── pom.xml
│
└── cotuba-core
    ├── pom.xml
    └── src
```

O módulo `cotuba-core` deve ser definido no supermódulo `cotuba`:

```
<!-- cotuba/pom.xml -->

<modules>
  <module>cotuba-core</module>
</modules>
```

Os seguintes pacotes precisariam ser movidos do diretório `cotuba-cli-backup/src/main/java` para o *source folder* `src/main/java` do módulo `cotuba-core`: `cotuba.application`, `cotuba.domain`, `cotuba.domain.builder`, `cotuba.md`, `cotuba.epub`, `cotuba.pdf`, `cotuba.html`, `cotuba.plugin`. Ou seja, apenas o pacote `cotuba.cli` seria deixado de fora.

É necessário definir como dependências as bibliotecas *CommonMark Java*, *iText pdfHTML* e *Epublib* no módulo `cotuba-core`:

```
<!-- cotuba/cotuba-core/pom.xml -->

<dependencies>
```

```
<dependency>
  <groupId>commons-cli</groupId>
  <artifactId>commons-cli</artifactId>
  <version>1.4</version>
</dependency>

<dependency>
  <groupId>com.atlassian.commonmark</groupId>
  <artifactId>commonmark</artifactId>
  <version>0.11.0</version>
</dependency>

<dependency>
  <groupId>nl.siegmann.epublib</groupId>
  <artifactId>epublib-core</artifactId>
  <version>3.1</version>
  <exclusions>
    <exclusion>
      <groupId>net.sf.kxml</groupId>
      <artifactId>kxml2</artifactId>
    </exclusion>
    <exclusion>
      <groupId>xmlpull</groupId>
      <artifactId>xmlpull</artifactId>
    </exclusion>
  </exclusions>
</dependency>

<dependency>
  <groupId>net.sf.kxml</groupId>
  <artifactId>kxml2</artifactId>
  <version>2.5.1-SNAPSHOT</version>
  <classifier>jar-with-dependencies</classifier>
  <exclusions>
    <exclusion>
      <groupId>org.xmlpull</groupId>
      <artifactId>xmlpull</artifactId>
    </exclusion>
  </exclusions>
</dependency>
```

```
  <dependency>
    <groupId>com.itextpdf</groupId>
    <artifactId>html2pdf</artifactId>
    <version>2.0.0</version>
    <exclusions>
      <exclusion>
        <groupId>com.itextpdf</groupId>
        <artifactId>forms</artifactId>
      </exclusion>
    </exclusions>
  </dependency>

</dependencies>
```

As bibliotecas iText pdfHTML e Epublib, assim como algumas de suas dependências, não estão definidas no repositório central do Maven. Precisam de repositórios próprios:

```
<!-- cotuba/cotuba-core/pom.xml -->

<repositories>

  <repository>
    <id>psiegman-repo</id>
    <url>https://github.com/psiegman/mvn-repo/raw/master/releases
      </url>
  </repository>

  <repository>
    <id>iText Repository</id>
    <name>iText Repository-releases</name>
    <url>https://repo.itextsupport.com/releases</url>
  </repository>

  <repository>
    <id>mvn-repo</id>
    <url>https://rawgit.com/alexandreaquiles/mvn-repo/master
      </url>
  </repository>
```

```
</repositories>
```

Após essas alterações, o módulo `cotuba-core` deve ser compilado com sucesso.

Criando um módulo para a linha de comando

Vamos criar o módulo `cotuba-cli`, que terá o código da interface de linha de comando, como um submódulo de `cotuba` com as seguintes configurações:

- o `<parent>` deve ser definido com `cotuba` tanto no `<groupId>` como no `<artifactId>` e com `0.0.1-SNAPSHOT` na `<version>`
- o `<artifactId>` do módulo deve ser definido como `cotuba-cli`

O módulo `cotuba-cli` teria um `pom.xml` parecido com:

```
<!-- cotuba/cotuba-cli/pom.xml -->

<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>cotuba</groupId>
    <artifactId>cotuba</artifactId>
    <version>0.0.1-SNAPSHOT</version>
  </parent>
  <artifactId>cotuba-cli</artifactId>

</project>
```

O diretório do módulo `cotuba-cli` deve ser definido dentro do diretório do supermódulo `cotuba`. A estrutura deve ser a seguinte:

```
cotuba
│
├── pom.xml
```

```
|
├── cotuba-core
|   ├── pom.xml
|   └── src
|
└── cotuba-cli
    ├── pom.xml
    └── src
```

No `pom.xml` do supermódulo, deve ser declarado o módulo `cotuba-cli`:

```
<!-- cotuba/pom.xml -->

<modules>
  <module>cotuba-core</module>
  <module>cotuba-cli</module> <!-- inserido -->
</modules>
```

No `pom.xml` do módulo `cotuba-cli`, deve ser definida a configuração do `maven-assembly-plugin` que define a criação do ZIP, que é o artefato disponibilizado para os usuários do Cotuba.

```
<!-- cotuba/cotuba-cli/pom.xml -->

<build>
  <plugins>
    <plugin>
      <artifactId>maven-assembly-plugin</artifactId>
      <version>3.1.0</version>
      <configuration>
        <descriptors>
          <descriptor>src/assembly/distribution.xml</descriptor>
        </descriptors>
      </configuration>
      <executions>
        <execution>
          <id>make-assembly</id>
          <phase>package</phase>
          <goals>
            <goal>single</goal>
```

```
          </goals>
        </execution>
      </executions>
    </plugin>
  </plugins>
</build>
```

O diretório `cotuba-cli-backup/src/assembly` deve ser copiado para dentro de `src` do módulo `cotuba-cli`. Nesse diretório, há o arquivo `distribution.xml`, que descreve a criação do ZIP do Cotuba. Também deve ser copiado para `cotuba-cli` o diretório `cotuba-cli-backup/src/scripts`, que contém o script `cotuba.sh`.

As dependências à biblioteca Apache Commons CLI e ao módulo `cotuba-core` devem ser declaradas no módulo `cotuba-cli`:

```
<!-- cotuba/cotuba-cli/pom.xml -->

<dependencies>

  <dependency>
    <groupId>commons-cli</groupId>
    <artifactId>commons-cli</artifactId>
    <version>1.4</version>
  </dependency>

  <dependency>
    <groupId>cotuba</groupId>
    <artifactId>cotuba-core</artifactId>
    <version>${project.version}</version>
  </dependency>

</dependencies>
```

As classes do pacote `cotuba.cli` devem ser copiadas para o *source folder* `src/main/java` de `cotuba-cli`.

Pronto! Podemos realizar o build do supermódulo `cotuba` com o seguinte comando:

```
# cotuba

mvn clean install
```

O resultado deve ser algo similar ao seguinte:

```
[INFO] ------------------------------------------------------------
[INFO] Reactor Summary for cotuba 0.0.1-SNAPSHOT:
[INFO]
[INFO] cotuba ............................. SUCCESS [  0.180 s]
[INFO] cotuba-core ........................ SUCCESS [  0.559 s]
[INFO] cotuba-cli ......................... SUCCESS [  1.464 s]
[INFO] ------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------
[INFO] Total time: 2.270 s
```

O diretório `cotuba-cli/livro-exemplo` pode ser movido para o diretório `cotuba`, o supermódulo. Para gerar um PDF, devemos usar o ZIP do `target` do módulo `cotuba-cli`:

```
unzip -o cotuba-cli/target/cotuba-*-distribution.zip -d ~/Desktop
cd ~/Desktop
./cotuba.sh -d solid-na-pratica/cotuba/livro-exemplo -f pdf
```

Se colocarmos módulos (JARs) de plugins como o `tema-paradizo` ou `estatisticas-ebook` no diretório `libs`, eles serão aplicados. Finalmente, podemos remover o diretório `cotuba-cli-backup`.

Atualizando os plugins de temas e estatísticas

Devemos compartilhar o módulo `cotuba-core` com as empresas Paradizo, que implementa o plugin de temas, e Cognitio, que implementa o plugin de cálculo de estatísticas. Como não estamos usando um repositório de artefatos como o Sonatype Nexus, esse compartilhamento deve ser feito pelo próprio repositório do GitHub.

Não é necessário, em ambos os plugins, depender do módulo de linha de comando. Basta que haja uma dependência ao módulo `cotuba-core`.

O time da Paradizo deve alterar, no projeto `tema-paradizo` , a dependência a `cotuba-cli` para `cotuba-core` em seu `pom.xml` :

```
<!-- tema-paradizo/pom.xml -->

<dependency>
  <groupId>cotuba</groupId>
  <artifactId>cotuba-cli</artifactId>-
  <artifactId>cotuba-core</artifactId>
  <version>0.0.1-SNAPSHOT</version>
</dependency>
```

O mesmo deve ser feito pelo time da Cognitio no `pom.xml` do projeto `estatisticas-ebook` .

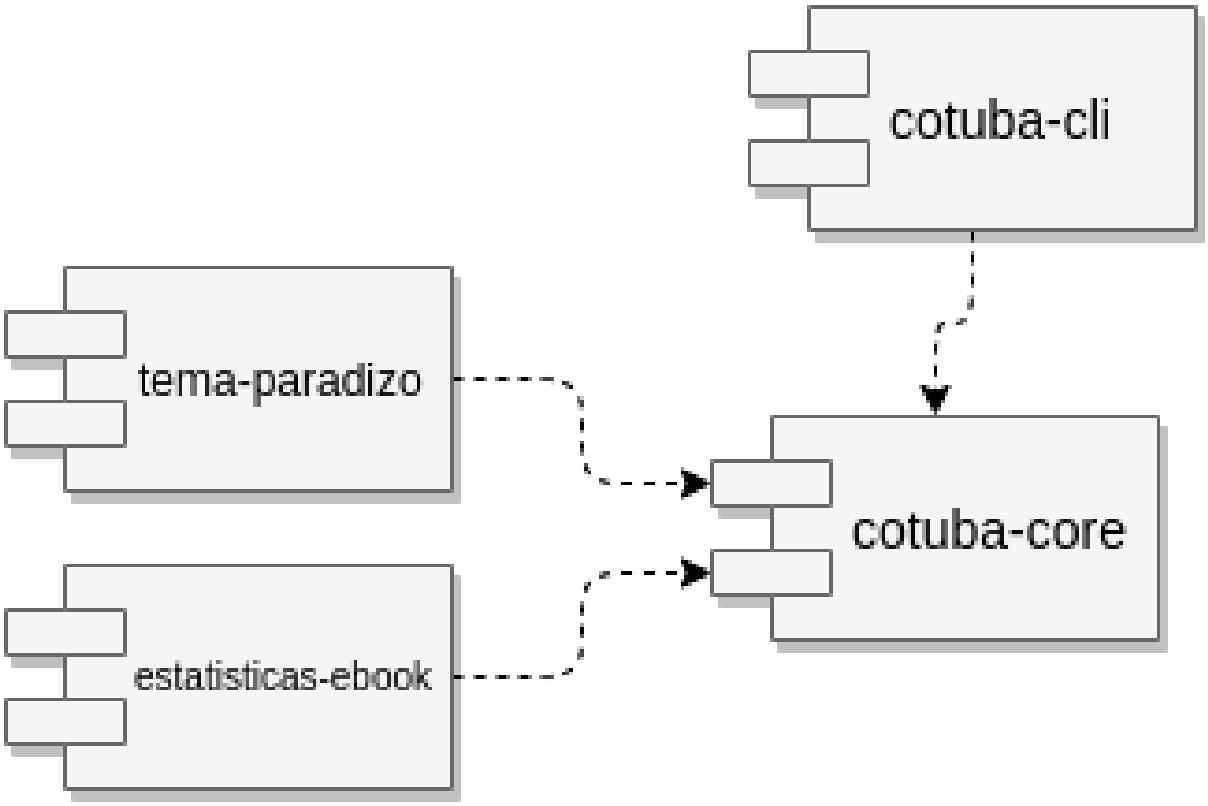

Figura 10.5: Cotuba modularizado e seus plugins

10.5 Princípios de acoplamento de módulos

Dificilmente uma aplicação será composta por apenas um módulo. Mesmo se o código da aplicação em si estiver modularizado, usaremos módulos (os JARs) de bibliotecas e frameworks. E esses módulos, por sua vez, comumente têm outros módulos como **dependência**. Podemos dizer, portanto, que (quase) todo módulo tem **acoplamento** com outros módulos.

É importante minimizar o acoplamento e ter o controle dessas dependências. Por isso, Uncle Bob, define alguns princípios relativos ao acoplamento de módulos.

O Princípio das Dependências Acíclicas (ADP)

Vamos voltar à ideia de módulos separados para a entrada e para a saída. Lembrando, teríamos os seguintes módulos:

- `cotuba-cli` , um módulo para a interface de linha de comando (a entrada)
- `cotuba-pdf` , `cotuba-epub` e `cotuba-html` , módulos para cada formato de ebook (as saídas)
- `cotuba-core` , um módulo para as regras de negócio.

As dependências entre os módulos seriam:

- `cotuba-cli` depende de `cotuba-core`
- `cotuba-pdf` , `cotuba-epub` e `cotuba-html` também dependem de `cotuba-core`
- `cotuba-core` , conforme mencionamos antes, depende das implementações dos geradores de ebook de `cotuba-pdf` , `cotuba-epub` e `cotuba-html` e `FormatoEbook`

Veja que haveria um ciclo nas dependências de `cotuba-core`, `cotuba-pdf` e `cotuba-epub`:

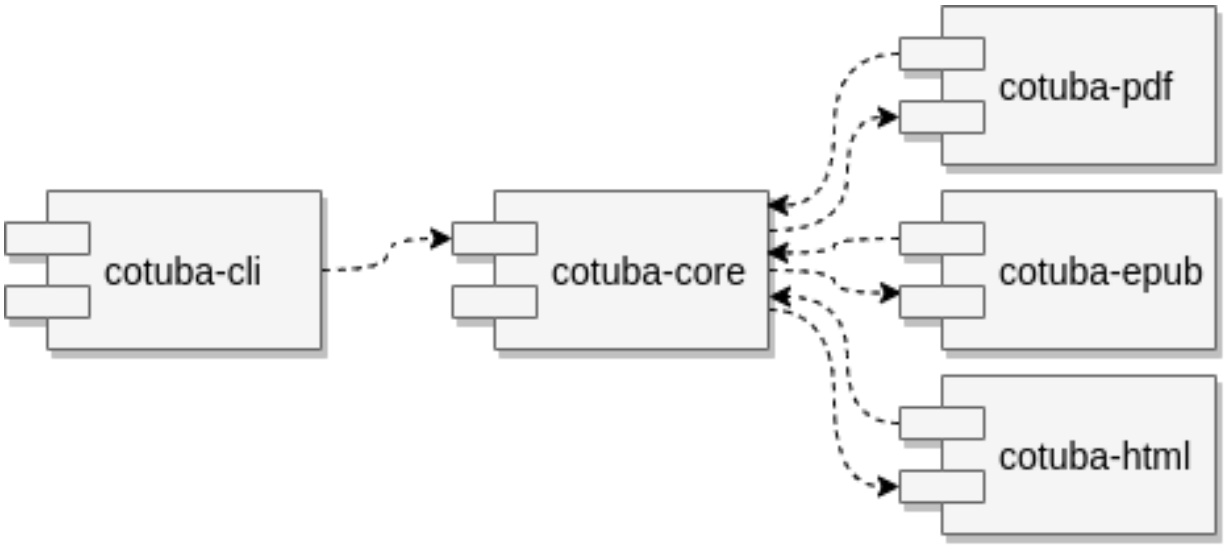

Figura 10.6: Dependência cíclica nos módulos do Cotuba

Se alguma aplicação desejar usar apenas o módulo `cotuba-pdf` para gerar PDFs, vai depender de `cotuba-core`. Até aí tudo bem!

Como `cotuba-core` depende tanto de `cotuba-pdf` como de `cotuba-epub` e `cotuba-html`, ao usarmos o gerador de PDFs, acabaríamos dependendo também do gerador de EPUBs. E isso ocorre por causa dessa **dependência cíclica**.

Às vezes, não é tão fácil descobrir as dependências cíclicas: pode ser que o ciclo se forme em dependências transitivas, as dependências das dependências. Por exemplo, do "ponto de vista" de `cotuba-cli`, que só depende diretamente de `cotuba-core`, não há ciclos. Mas há uma dependência transitiva, por meio de `cotuba-core`, com `cotuba-pdf`, `cotuba-epub` e `cotuba-html`. E esses módulos formam o ciclo que destacamos anteriormente.

Dependências cíclicas também afetam a compilação: para compilar `cotuba-core`, precisaríamos do JAR de `cotuba-pdf`, `cotuba-epub` e `cotuba-html`. Porém, para compilar `cotuba-pdf`, por exemplo,

precisaríamos do JAR de `cotuba-core`. Um dilema. Um beco sem saída.

Dependências cíclicas são tão problemáticas que um projeto Maven multimódulos não pode ter esse tipo de acoplamento. Se tiver, o projeto não será compilado com sucesso.

Para reafirmar o problema de dependências cíclicas, Uncle Bob cunhou o seguinte princípio:

> **Acyclic Dependencies Principle (ADP)**
>
> *Não permita ciclos no grafo de dependências de módulos.*

No clássico artigo *On the Criteria To Be Used in Decomposing Systems into Modules* (PARNAS, 1972), David Parnas menciona que uma característica desejável para a estrutura de um sistema é ter uma relação hierárquica entre os módulos, apresentando um conjunto parcialmente ordenado. Conjuntos parcialmente ordenados são grafos acíclicos direcionados, ou seja, não possuem ciclos. Parnas conclui afirmando que relações hierárquicas são características desejáveis, mas independentes de decomposições que usam seu critério de *information hiding*. É importante dizer que, no artigo de Parnas, *"módulo é considerado uma atribuição de responsabilidade em vez de um subprograma.*"

Quebrando ciclos com abstrações

E como quebramos ciclos? Devemos inverter as dependências! O módulo `cotuba-core` poderia fornecer a abstração `GeradorEbook`, que seria implementada pelas classes:

- `GeradorPDF`, do módulo `cotuba-pdf`
- `GeradorEPUB`, do módulo `cotuba-epub`
- `GeradorHTML`, do módulo `cotuba-html`.

Todos os módulos de geração de ebook passariam a depender de `cotuba-core`. Não o contrário.

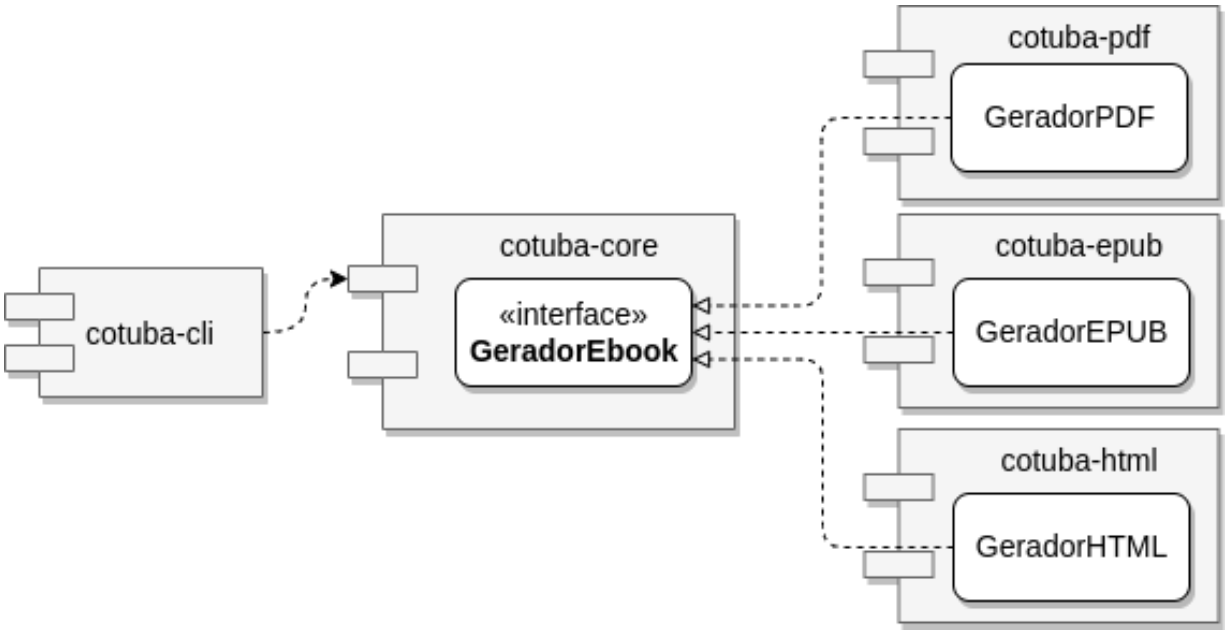

Figura 10.7: Invertendo dependências para quebrar ciclos

Poderíamos transformar os módulos de geradores de ebook em plugins. A interface `GeradorEbook` seria a SPI. As implementações dos geradores para cada formato de ebook seriam *service providers*. Obteríamos as implementações disponíveis para essa SPI através da classe `ServiceLoader`. Dessa forma, a enum `FormatoEbook` deixaria de depender das implementações.

Um efeito colateral da quebra dos ciclos no Cotuba é que `cotuba-cli` não precisaria depender indiretamente dos geradores de ebooks, já que não são mais dependências transitivas de `cotuba-core`. Isso encaminha o design do código na direção do CRP, já que evitaríamos dependências desnecessárias.

O Princípio das Dependências Estáveis (SDP)

Para Uncle Bob, a **estabilidade** de um módulo está relacionada com o cuidado necessário para mudá-lo. Um módulo muito estável é aquele em que temos que pensar bastante antes de fazer qualquer

mudança. E quando isso acontece? Quando há outros módulos que dependem de um dado módulo. Quanto mais módulos dependerem desse módulo, maior é sua estabilidade.

Um detalhe interessante: em português, é comum que a palavra "estabilidade" tenha uma conotação positiva. Porém, ao pensarmos em módulos, estabilidade é apenas uma característica, sem juízo de valor. Módulos estáveis são aqueles que devemos modificar com muito cuidado.

Para ter uma ideia da estabilidade de um módulo, devemos contar quantos outros módulos dependem dele. Em um diagrama, seriam o número de setas que entram em um módulo. Considerando que temos módulos separados para cada gerador de ebook, vamos comparar a estabilidade de dois possíveis designs de módulos:

- `cotuba-cli` depende de `cotuba-core` que depende de `cotuba-pdf`, `cotuba-epub` e `cotuba-html` que, por sua vez, dependem de `cotuba-core`.
- `cotuba-cli` depende de `cotuba-core` que, seguindo o conceito de inversão de dependências, teria uma abstração de gerador de ebook que seria implementada pelos módulos `cotuba-pdf`, `cotuba-epub` e `cotuba-html`.

Analisando a estabilidade do primeiro design, em que há uma dependência direta do módulo `cotuba-core` aos geradores de ebook:

- nenhum módulo depende de `cotuba-cli` e, portanto, não há exigência de estabilidade para essa módulo;
- o módulo de linha de comando depende do módulo `cotuba-core`, o que exige estabilidade;
- o módulo core depende de `cotuba-pdf`, `cotuba-epub` e `cotuba-html` e, então, há exigência de estabilidade para os módulos geradores de ebook.

A necessidade de estabilidade dificulta a mudança. Por exemplo, uma mudança no módulo `cotuba-pdf` afetaria `cotuba-core` e,

indiretamente, `cotuba-cli`. Já, quando analisamos o segundo design, que quebra ciclos por meio da inversão de dependências:

- não há exigência de estabilidade para `cotuba-cli`;
- também não temos preocupações com a estabilidade dos módulos geradores de ebook `cotuba-pdf`, `cotuba-epub` e `cotuba-html`;
- há uma exigência maior de estabilidade para `cotuba-core` porque teríamos os outros 4 módulos dependendo desse.

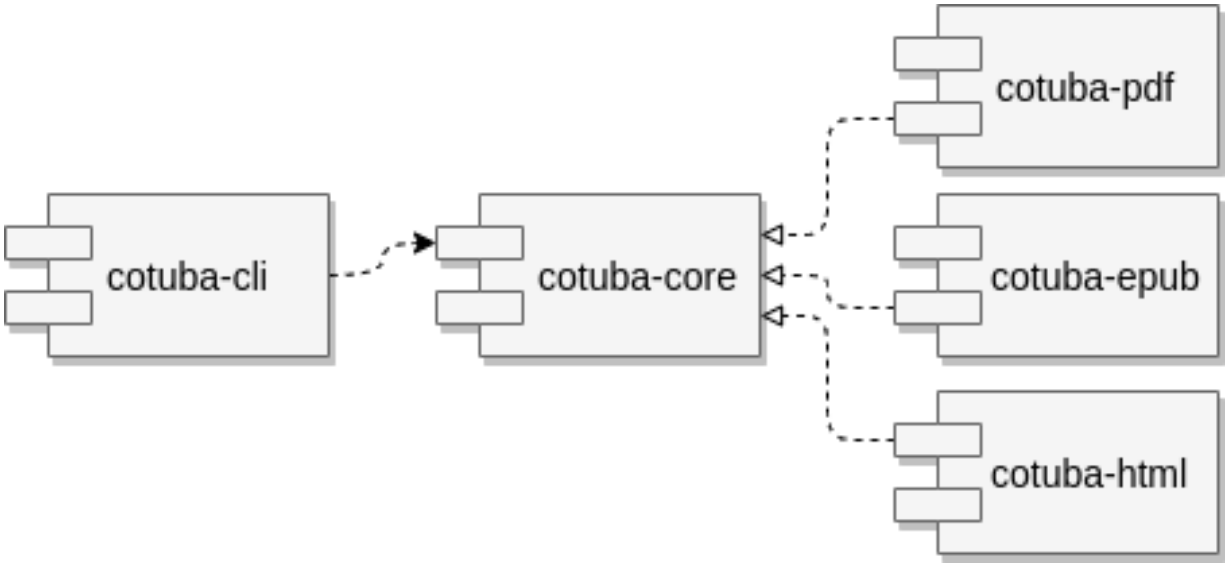

Figura 10.8: Dependências na direção do módulo mais estável

Perceba, no diagrama anterior, que as setas que representam as dependências apontam para o mesmo módulo central, o `cotuba-core`. Há uma grande exigência de estabilidade para o módulo central. Os módulos periféricos não precisam ser estáveis, já que não há outros módulos que dependem deles. Podemos dizer que as setas estão direcionadas ao módulo mais estável.

Uncle Bob consideraria esse último design melhor pois, considerando o todo, temos uma menor quantidade de módulos com exigência de estabilidade. A mudança ficaria facilitada. A ideia de ter as dependências entre módulos direcionadas para o módulo mais estável é descrita no seguinte princípio:

Stable Dependencies Principle (SDP)

Tenha dependências na direção da estabilidade.

É interessante notar que, ao seguirmos o ADP e utilizarmos abstrações para quebrar o ciclo de dependências do Cotuba, passamos a caminhar na direção do SDP. Ambos ao mesmo tempo!

Métrica: instabilidade

Uncle Bob define uma métrica que considera o número de dependências que entram e que saem de um módulo: a **instabilidade** (ou *instability*, em inglês). Para calcular a instabilidade de um módulo, precisamos da seguinte métrica:

- *fan-in*: conta as dependências de entrada, ou seja, o número de classes externas, de outros módulos, que dependem de classes internas de um módulo.
- *fan-out*: conta as dependências de saída, ou seja, o número de classes internas de um módulo que dependem de classes externas, de outros módulos.

Para contar esses números de classes externas, devemos analisar os imports de cada classe, identificando quais ultrapassam a barreira do módulo. A partir dessas informações podemos calcular a instabilidade `I` com a seguinte fórmula:

```
I = fan-out / (fan-in + fan-out)
```

O valor de `I` será entre 0 e 1. Um valor 0 indica que um módulo não depende de nenhum outro (fan-out é 0), mas outros módulos o têm como dependência (fan-in maior que 0). Tal módulo apresentaria o máximo de estabilidade possível.

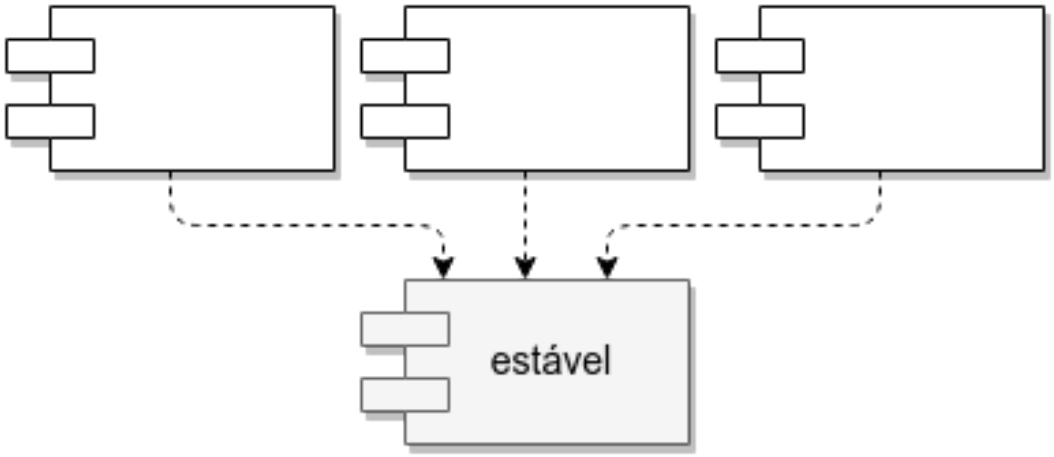

Figura 10.9: Módulo totalmente estável

Um valor 1 indica que um módulo depende de vários outros (fan-out maior que 0) mas nenhum módulo o tem como dependência (fan-in é 0). Tal módulo apresentaria o máximo de instabilidade possível.

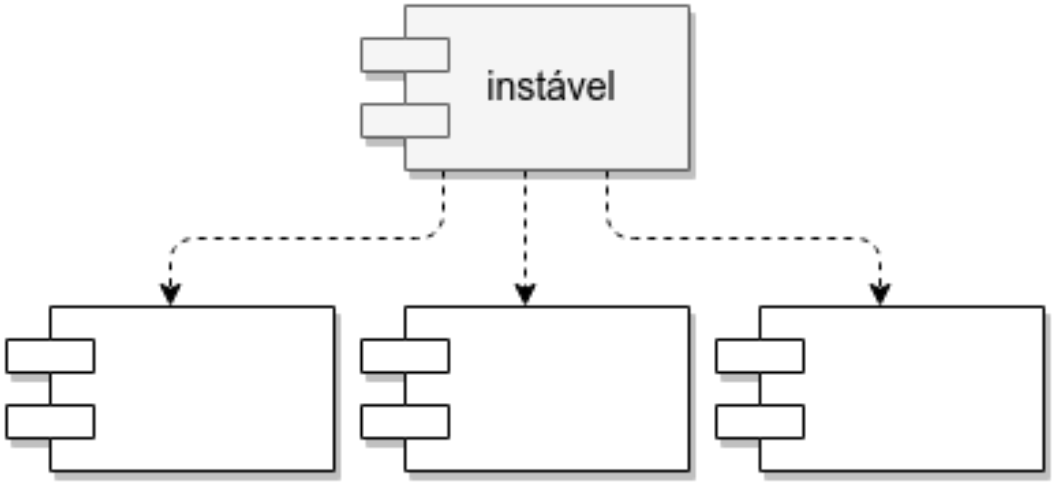

Figura 10.10: Módulo totalmente instável

De acordo com o SDP, um módulo deve depender de módulos mais estáveis. Portanto, a métrica `I` de um módulo deve ser maior que as métricas dos módulos dos quais ele depende.

O Princípio das Abstrações Estáveis (SAP)

Módulos em que há a exigência de estabilidade precisam ser mudados com cuidado: uma pequena alteração pode exigir modificações em vários outros módulos. Porém, podemos tornar esses módulos estáveis fáceis de estender fazendo com que eles definam abstrações. Abstrações levam à flexibilidade. Quanto mais interfaces e classes abstratas forem definidas em um módulo, mais fácil será estendê-lo.

Um módulo para o qual há uma grande exigência de estabilidade deve ser composto, em grande parte, por abstrações. Uncle Bob define um princípio com esse conceito:

> **Stable Abstractions Principle (SAP)**
>
> *Um módulo deve ser tão abstrato quanto for estável.*

Uncle Bob compara o SDP, SAP e DIP no livro *Agile Principles, Patterns, and Practices in C#* (MARTIN, 2006): *"Combinados, o SDP e o SAP formam o DIP para (módulos). O SDP diz que as dependências devem ser na direção da estabilidade. O SAP diz que estabilidade implica em abstração. Portanto, as dependências devem ser na direção da abstração. Entretanto, o DIP lida com classes. Com classes, não há tons de cinza. Uma classe é abstrata ou não é. A combinação do SDP com o SAP lida com (módulos) e permite que um (módulo) seja parcialmente abstrato ou parcialmente estável."*

Métrica: abstratividade

Uncle Bob também define uma métrica que considera a proporção de abstrações de um módulo: a **abstratividade** (ou *abstractness*, em inglês). Considere que Na é o número de classes abstratas e interfaces de um módulo e Nc é o número total de tipos, somando classes e interfaces. O cálculo da abstratividade A deve ser feito com a fórmula a seguir:

```
A = Na / Nc
```

O valor de `A` varia de 0 a 1. Um valor 0 indica que o módulo não possui nenhuma classe abstrata nem interface. Um valor 1 indica que o módulo tem somente classes abstratas ou interfaces.

De acordo com o SAP, a abstratividade deve acompanhar a estabilidade de um módulo. Por isso, podemos dizer que quanto menor a métrica `I` maior deverá ser a abstratividade `A`.

Para saber mais: API Modules

No design sugerido anteriormente, em que o módulo `cotuba-core` fornece uma abstração `GeradorEbook` para ser implementada pelos módulos `cotuba-pdf`, `cotuba-epub` e `cotuba-html` há um problema: os módulos das implementações de geradores de ebook podem depender de detalhes internos do Cotuba, não apenas da abstração.

A abstratividade `A` do módulo `cotuba-core` é baixa. E como há muitas dependências que chegam nesse módulo, sua instabilidade `I` também é baixa. Esse não é o cenário ideal, de acordo com os princípios de componente de Uncle Bob.

Uma maneira de atender ao SDP e ao SAP, fazendo com que o módulo de menor instabilidade `I` (o mais estável) tenha alta abstratividade `A`, é separar a abstração `GeradorEbook` em um módulo próprio. Poderíamos chamar esse módulo de `cotuba-ebooks`, por exemplo. Teríamos dentro dele apenas a interface necessária, talvez adicionando abstrações para o domain model.

Poderíamos fazer algo semelhante com as abstrações de plugins: as SPIs `AoRenderizarHTML` e `AoFinalizarGeracao`. Com um módulo `cotuba-plugin` com alta abstratividade, os *service providers* que as implementam não teriam acesso a detalhes indesejados do Cotuba.

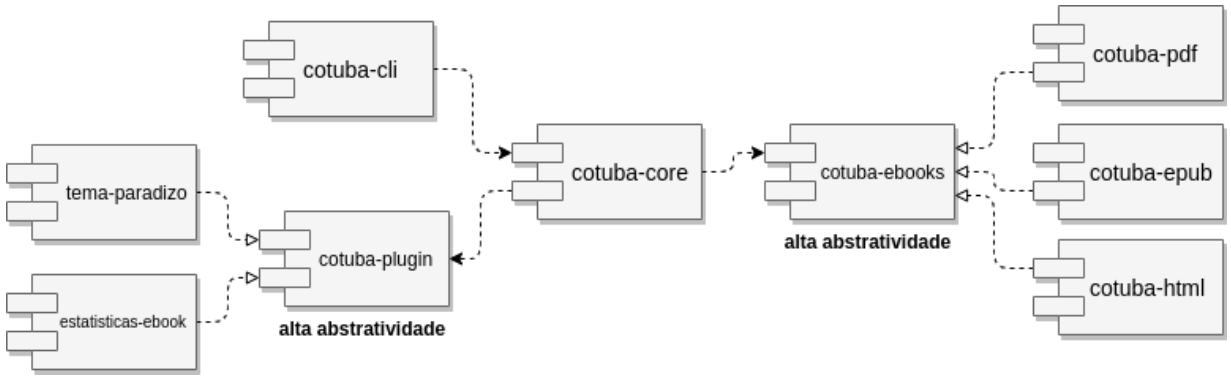

Figura 10.11: Módulos de API

Perceba na imagem anterior que os módulos mais estáveis, com mais setas que chegam, seriam os módulos com alta abstratividade. Módulos que contêm apenas abstrações são *modularity patterns* recomendados por diferentes autores.

Kirk Knoernschild, no livro *Java Application Architecture: Modularity Patterns* (KNOERNSCHILD, 2012), chama de *Separate Abstractions* (Abstrações Separadas) um pattern semelhante: *"Coloque abstrações e as classes que as implementam em módulos separados."*

No livro *Java 9 Modularity* (MAK; BAKKER, 2017), os autores Sander Mak e Paul Bakker chamam o pattern de *API Module* (Módulo de API): *"Aplicações que consistem de módulos que exportam tudo o que contêm são normalmente um sinal de alerta, não nos deixando em situação melhor do que antes de usar módulos. Uma aplicação modularizada oculta detalhes de implementação de outras partes da aplicação, assim como uma boa biblioteca oculta seus componentes internos das aplicações. Sempre que seu módulo for usado em diferentes partes de sua aplicação, ou por diferentes equipes em sua organização, ter uma API bem definida e estável é fundamental."*

Não definiremos API Modules neste livro porque seria uma solução desnecessariamente complexa. Veremos maneiras diferentes de limitar o que é exposto por um módulo em um capítulo posterior.

10.6 Quebrando ciclos no Cotuba

Vamos implementar a quebra dos ciclos do Cotuba criando os seguintes submódulos de `cotuba`: `cotuba-pdf`; `cotuba-epub` e `cotuba-html`.

Esses módulos de geração de ebooks devem ter como dependência o módulo `cotuba-core` que, por sua vez, terá a interface `GeradorEbook` como SPI.

Um módulo para a geração de PDFs

Começaremos criando um módulo `cotuba-pdf`, que contém o código para a geração de PDFs como submódulo de `cotuba`. O novo módulo deve ter as seguintes configurações:

- o `<parent>` deve ser definido com `cotuba` tanto no `<groupId>` como no `<artifactId>` e com `0.0.1-SNAPSHOT` na `<version>`;
- o `<artifactId>` do módulo deve ser definido como `cotuba-pdf`.

O `pom.xml` do módulo `cotuba-pdf` seria algo como:

```
<!-- cotuba/cotuba-pdf/pom.xml -->

<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>cotuba</groupId>
    <artifactId>cotuba</artifactId>
    <version>0.0.1-SNAPSHOT</version>
  </parent>
  <artifactId>cotuba-pdf</artifactId>
```

```
</project>
```

A estrutura de diretórios ficaria semelhante à seguinte:

```
cotuba
│
├── pom.xml
│
├── cotuba-core
│   ├── pom.xml
│   └── src
│
├── cotuba-cli
│   ├── pom.xml
│   └── src
│
└── cotuba-pdf
    ├── pom.xml
    └── src
```

O novo módulo deve ser definido no `pom.xml` do supermódulo `cotuba`:

```
<!-- cotuba/pom.xml -->

<modules>
  <module>cotuba-cli</module>
  <module>cotuba-core</module>
  <module>cotuba-pdf</module> <!-- inserido -->
</modules>
```

O pacote `cotuba.pdf` deve ser **movido** do módulo `cotuba-core` para o *source folder* `src/main/java` do módulo `cotuba-pdf`. Devem acontecer alguns erros de compilação em ambos os módulos.

Devem ser declaradas como dependências de `cotuba-pdf` a biblioteca *iText pdfHTML* e o módulo `cotuba-core`:

```
<!-- cotuba/cotuba-pdf/pom.xml -->
```

```
<dependencies>

  <dependency>
    <groupId>cotuba</groupId>
    <artifactId>cotuba-core</artifactId>
    <version>${project.version}</version>
  </dependency>

  <dependency>
    <groupId>com.itextpdf</groupId>
    <artifactId>html2pdf</artifactId>
    <version>2.0.0</version>
    <exclusions>
      <exclusion>
        <groupId>com.itextpdf</groupId>
        <artifactId>forms</artifactId>
      </exclusion>
    </exclusions>
  </dependency>

</dependencies>
```

A configuração do repositório de artefatos do iText pdfHTML deve ser **movida** de `cotuba-core` para `cotuba-pdf`:

```
<!-- cotuba/cotuba-pdf/pom.xml -->

<repositories>

  <repository>
    <id>iText Repository</id>
    <name>iText Repository-releases</name>
    <url>https://repo.itextsupport.com/releases</url>
  </repository>

</repositories>
```

Nesse momento, a enum `FormatoEbook` do módulo `cotuba-core` apresenta erros de compilação relacionados ao gerador de PDFs:

```
// cotuba/cotuba-core
// cotuba.domain.FormatoEbook

import cotuba.epub.GeradorEPUB;
import cotuba.pdf.GeradorPDF; // error: package does not exist
import cotuba.html.GeradorHTML;

public enum FormatoEbook {

  PDF(new GeradorPDF()), // error: cannot find symbol
  EPUB(new GeradorEPUB()),
  HTML(new GeradorHTML());

  // restante do código...

}
```

Uma ideia seria resolver esse erro declarando `cotuba-pdf` como dependência de `cotuba-core`:

```
<!-- cotuba/cotuba-core/pom.xml -->

<dependency>
  <groupId>cotuba</groupId>
  <artifactId>cotuba-pdf</artifactId>
  <version>${project.version}</version>
</dependency>
```

Porém, ao fazermos isso, estaríamos criando um ciclo entre os módulos: `cotuba-pdf` depende de `cotuba-core` que, por sua vez, passaria a depender de `cotuba-pdf`. Poderíamos tentar fazer o build do Cotuba com o comando:

```
# cotuba

mvn clean install
```

Como resultado, seria exibido um erro de referência cíclica:

```
[ERROR] The projects in the reactor contain a cyclic reference:
Edge between 'Vertex{label='cotuba:cotuba-pdf:0.0.1-SNAPSHOT'}'
```

```
    and 'Vertex{label='cotuba:cotuba-core:0.0.1-SNAPSHOT'}'
introduces to cycle in the graph
    cotuba:cotuba-core:0.0.1-SNAPSHOT -->
    cotuba:cotuba-pdf:0.0.1-SNAPSHOT -->
    cotuba:cotuba-core:0.0.1-SNAPSHOT @
```

Perceba que o *reactor*, o mecanismo do Maven que lida com projetos multimódulos, detecta o ciclo e impede a compilação do projeto!

Para saber mais: detectando ciclos com ArchUnit

Mesmo em um projeto que não usa módulos Maven, é possível detectar ciclos usando ferramentas de análise de código como o ArchUnit. O primeiro passo é definir o ArchUnit como dependência do nosso projeto. A versão do ArchUnit compatível com o JUnit 5 é a seguinte:

```
<dependency>
  <groupId>com.tngtech.archunit</groupId>
  <artifactId>archunit-junit5</artifactId>
  <version>0.21.0</version>
  <scope>test</scope>
</dependency>
```

Em seguida, devemos criar uma classe anotada com `@AnalyzeClasses`, passando o pacote que deve ser analisado como parâmetro. Dentro dessa classe, devemos definir atributos do tipo `ArchRule` e anotá-los com `@ArchTest`. É gerado um teste automatizado compatível com o JUnit que falha no caso de a regra ser violada. Podemos definir que os subpacotes diretos de Cotuba não podem ter ciclos, da seguinte maneira:

```
@AnalyzeClasses(packages = "cotuba")
class ModularityTest {

  @ArchTest
  static final ArchRule no_cycles =
      slices().matching("cotuba.(*)..")
          .should()
```

```
            .beFreeOfCycles();

}
```

Se executarmos a regra anterior do ArchUnit com o código do Cotuba antes de iniciarmos a modularização deste capítulo, o teste falharia com diversas violações:

```
java.lang.AssertionError:
  Architecture Violation [Priority: MEDIUM] -
Rule 'slices matching 'cotuba.(*)..' should be free of cycles'
    was violated (19 times):
...
Cycle detected: Slice domain ->
                Slice pdf ->
                Slice domain
  1. Dependencies of Slice domain
    - Static Initializer <cotuba.domain.FormatoEbook.<clinit>()>
        calls constructor <cotuba.pdf.GeradorPDF.<init>()>
        in (FormatoEbook.java:10)
  2. Dependencies of Slice pdf
    - Method <cotuba.pdf.GeradorPDF.gera(cotuba.domain.Ebook)>
        has parameter of type <cotuba.domain.Ebook>
        in (GeradorPDF.java:0)
...
```

A violação anterior indica que tanto o pacote `cotuba.domain` depende do pacote `cotuba.pdf` como o contrário. A mensagem de erro aponta que:

- o construtor de `GeradorPDF`, de `cotuba.pdf`, é usado em `FormatoEbook`, de `cotuba.domain`;
- a classe `Ebook`, de `cotuba.domain`, é usada como parâmetro em um método de `GeradorPDF`, de `cotuba.pdf`.

É possível definir diversas outras regras com o ArchUnit, como reforçar a organização em camadas e outros estilos arquiteturais. Mais informações podem ser encontradas na documentação oficial: https://www.archunit.org/

O código anterior pode ser encontrado em: https://github.com/alexandreaquiles/solid-na-pratica/tree/cap10-extra-arch-unit

Quebrando ciclos através de plugins

Teremos que voltar atrás na decisão que adicionou uma dependência cíclica entre os módulos do Cotuba. Vamos remover do `pom.xml` de `cotuba-core` a dependência a `cotuba-pdf`:

```
<!-- cotuba/cotuba-core/pom.xml -->

<dependency>
  <groupId>cotuba</groupId>
  <artifactId>cotuba-pdf</artifactId>
  <version>${project.version}</version>
</dependency>
```

Vamos transformar a interface `GeradorEbook` em uma SPI. Para que seja possível saber qual o formato de ebook do gerador, vamos criar um método `formato` nessa interface, que retorna um `FormatoEbook`.

Além disso, vamos organizar os pacotes, movendo a interface `GeradorEbook` do pacote `cotuba.application` para o pacote `cotuba.plugin`, ainda do mesmo módulo `cotuba-core`. O método `cria` deve ser modificado para obter a implementação do gerador de um determinado formato a partir do `ServiceLoader`.

```
// cotuba/cotuba-core
// cotuba.plugin.GeradorEbook - movida!

public interface GeradorEbook {

  void gera(Ebook ebook);

  FormatoEbook formato(); // inserido

  // modificado
  static GeradorEbook cria(FormatoEbook formato) {
```

```
    for (GeradorEbook gerador :
        ServiceLoader.load(GeradorEbook.class)) {
      if (gerador.formato().equals(formato)) {
        return gerador;
      }
    }

    throw new IllegalArgumentException(
        "Formato do ebook inválido: " + formato);
  }

}
```

A enum `FormatoEbook`, do pacote `cotuba.domain` do módulo `cotuba-core`, deve ser simplificada para que não haja dependências às classes `GeradorPDF`, `GeradorEPUB` e `GeradorHTML`. Devemos deixar somente os valores:

```
// cotuba/cotuba-core
// cotuba.domain.FormatoEbook

// imports removidos!

public enum FormatoEbook {
  PDF, EPUB, HTML;
}
```

A classe `GeradorPDF`, do pacote `cotuba.pdf` do módulo `cotuba-pdf`, deve definir o método `formato` com o valor apropriado:

```
// cotuba/cotuba-pdf
// cotuba.pdf.GeradorPDF

import cotuba.domain.FormatoEbook; // inserido

public class GeradorPDF implements GeradorEbook {

  // inserido
  @Override
  public FormatoEbook formato() {
    return FormatoEbook.PDF;
```

```
  }

  // código omitido...

}
```

Ainda no módulo `cotuba-pdf`, deve ser definido um arquivo com o nome da SPI cujo conteúdo é o nome do *service provider*:

```
cotuba/cotuba-pdf/src/main/resources/META-
INF/services/cotuba.plugin.GeradorEbook

cotuba.pdf.GeradorPDF
```

Também devemos definir o método `formato` na classe `GeradorEPUB`, que ainda está no módulo `cotuba-core`:

```
// cotuba/cotuba-core
// cotuba.epub.GeradorEPUB

import cotuba.domain.FormatoEbook; // inserido

public class GeradorEPUB implements GeradorEbook {

  // inserido
  @Override
  public FormatoEbook formato() {
    return FormatoEbook.EPUB;
  }

  // código omitido...

}
```

A declaração do método `formato` também deve ser realizada para `GeradorHTML`, que também continua no `cotuba-core`:

```
// cotuba/cotuba-core
// cotuba.html.GeradorHTML

import cotuba.domain.FormatoEbook; // inserido
```

```
public class GeradorHTML implements GeradorEbook {

  // inserido
  @Override
  public FormatoEbook formato() {
    return FormatoEbook.HTML;
  }

  // código omitido...

}
```

Abra o Terminal e faça o build do Cotuba:

```
# cotuba

mvn clean install
```

Com os ciclos quebrados por meio de plugins, a compilação terá sucesso:

```
[INFO] ------------------------------------------------------------
[INFO] Reactor Summary for cotuba 0.0.1-SNAPSHOT:
[INFO]
[INFO] cotuba ............................. SUCCESS [  0.086 s]
[INFO] cotuba-core ........................ SUCCESS [  1.177 s]
[INFO] cotuba-cli ......................... SUCCESS [  0.547 s]
[INFO] cotuba-pdf ......................... SUCCESS [  0.151 s]
[INFO] ------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------
[INFO] Total time: 2.032 s
```

Dependência cíclica removida! ADP atendido!

Um módulo para a geração de EPUBs

Vamos extrair o código de geração de EPUBs do `cotuba-core` para um novo módulo `cotuba-epub` com as seguintes configurações:

- o `<parent>` deve ser definido com `cotuba` tanto no `<groupId>` como no `<artifactId>` e com `0.0.1-SNAPSHOT` na `<version>`;
- o `<artifactId>` do módulo deve ser definido como `cotuba-epub`.

O módulo `cotuba-epub` teria um `pom.xml` parecido com:

```xml
<!-- cotuba/cotuba-epub/pom.xml -->

<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>cotuba</groupId>
    <artifactId>cotuba</artifactId>
    <version>0.0.1-SNAPSHOT</version>
  </parent>
  <artifactId>cotuba-epub</artifactId>

</project>
```

Teríamos a seguinte estrutura de diretórios:

```
cotuba
|
├── pom.xml
|
├── cotuba-core
|   ├── pom.xml
|   └── src
|
├── cotuba-cli
|   ├── pom.xml
|   └── src
|
├── cotuba-pdf
|   ├── pom.xml
|   └── src
|
```

```
└── cotuba-epub
    ├── pom.xml
    └── src
```

No supermódulo `cotuba`, deve ser declarado o novo módulo:

```
<!-- cotuba/pom.xml -->

<modules>
  <module>cotuba-cli</module>
  <module>cotuba-core</module>
  <module>cotuba-pdf</module>
  <module>cotuba-epub</module> <!-- inserido -->
</modules>
```

Devemos **mover** o conteúdo do pacote `cotuba.epub` do módulo `cotuba-core` para o *source folder* `src/main/java` do módulo `cotuba-epub`.

Para corrigir os erros de compilação, devem ser declaradas como dependências a biblioteca Epublib e o módulo `cotuba-core`. Também deve ser adicionada a biblioteca kXML, uma dependência de Epublib:

```
<!-- cotuba/cotuba-epub/pom.xml -->

<dependencies>

  <dependency>
    <groupId>cotuba</groupId>
    <artifactId>cotuba-core</artifactId>
    <version>${project.version}</version>
  </dependency>

  <dependency>
    <groupId>nl.siegmann.epublib</groupId>
    <artifactId>epublib-core</artifactId>
    <version>3.1</version>
    <exclusions>
      <exclusion>
        <groupId>net.sf.kxml</groupId>
```

```
        <artifactId>kxml2</artifactId>
      </exclusion>
      <exclusion>
        <groupId>xmlpull</groupId>
        <artifactId>xmlpull</artifactId>
      </exclusion>
    </exclusions>
  </dependency>

  <dependency>
    <groupId>net.sf.kxml</groupId>
    <artifactId>kxml2</artifactId>
    <version>2.5.1-SNAPSHOT</version>
    <classifier>jar-with-dependencies</classifier>
    <exclusions>
      <exclusion>
        <groupId>org.xmlpull</groupId>
        <artifactId>xmlpull</artifactId>
      </exclusion>
    </exclusions>
  </dependency>

</dependencies>
```

Os repositórios de artefatos das bibliotecas Epublib e kXML devem ser **movidos** de `cotuba-core` para `cotuba-epub` :

```
<!-- cotuba/cotuba-epub/pom.xml -->

<repositories>

  <repository>
    <id>psiegman-repo</id>
    <url>https://github.com/psiegman/mvn-repo/raw/master/releases
    </url>
  </repository>

  <repository>
    <id>mvn-repo</id>
    <url>https://rawgit.com/alexandreaquiles/mvn-repo/master
    </url>
```

```
  </repository>

</repositories>
```

No módulo `cotuba-epub`, defina o arquivo com o nome da SPI de geração de ebooks cujo conteúdo é o nome do *service provider*:

`cotuba/cotuba-epub/src/main/resources/META-INF/services/cotuba.plugin.GeradorEbook`

```
cotuba.epub.GeradorEPUB
```

Um módulo para a geração de HTMLs

Finalmente, devemos criar um novo módulo `cotuba-html` que conterá o código de geração de HTML e deve ter as seguintes configurações:

- o `<parent>` deve ser definido com `cotuba` tanto no `<groupId>` como no `<artifactId>`, e com `0.0.1-SNAPSHOT` na `<version>`;
- o `<artifactId>` do módulo deve ser definido como `cotuba-html`.

O `pom.xml` do módulo `cotuba-html` será semelhante ao seguinte:

```
<!-- cotuba/cotuba-html/pom.xml -->

<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
  http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>cotuba</groupId>
    <artifactId>cotuba</artifactId>
    <version>0.0.1-SNAPSHOT</version>
  </parent>
  <artifactId>cotuba-html</artifactId>

</project>
```

Teríamos a seguinte estrutura de diretórios:

```
cotuba
│
├── pom.xml
│
├── cotuba-core
│   ├── pom.xml
│   └── src
│
├── cotuba-cli
│   ├── pom.xml
│   └── src
│
├── cotuba-pdf
│   ├── pom.xml
│   └── src
│
├── cotuba-epub
│   ├── pom.xml
│   └── src
│
└── cotuba-html
    ├── pom.xml
    └── src
```

O novo módulo `cotuba-html` deve ser declarado no supermódulo `cotuba`:

```
<!-- cotuba/pom.xml -->

<modules>
  <module>cotuba-cli</module>
  <module>cotuba-core</module>
  <module>cotuba-pdf</module>
  <module>cotuba-epub</module>
  <module>cotuba-html</module> <!-- inserido -->
</modules>
```

O conteúdo do pacote `cotuba.html` deve ser **movido** do módulo `cotuba-core` para o *source folder* `src/main/java` do módulo `cotuba-`

`html` . O módulo `cotuba-core` deve ser declarado como dependência:

```
<!-- cotuba/cotuba-html/pom.xml -->

<dependencies>

  <dependency>
    <groupId>cotuba</groupId>
    <artifactId>cotuba-core</artifactId>
    <version>${project.version}</version>
  </dependency>

</dependencies>
```

Devemos, também, definir o arquivo com o nome da SPI de geração de ebooks com o nome do *service provider* como conteúdo:

```
cotuba/cotuba-html/src/main/resources/META-
INF/services/cotuba.plugin.GeradorEbook

cotuba.html.GeradorHTML
```

Ufa! Acabamos de transformar os geradores de ebooks nos formatos PDF, EPUB e HTML em plugins, usando a interface `GeradorEbook` como SPI. Agora, podemos realizar o build do Cotuba com o comando:

```
# cotuba

mvn clean install
```

O resultado deverá ser algo como:

```
[INFO] ------------------------------------------------------------
[INFO] Reactor Summary for cotuba 0.0.1-SNAPSHOT:
[INFO]
[INFO] cotuba ............................. SUCCESS [  0.153 s]
[INFO] cotuba-core ........................ SUCCESS [  1.093 s]
[INFO] cotuba-cli ......................... SUCCESS [  0.527 s]
[INFO] cotuba-pdf ......................... SUCCESS [  0.223 s]
[INFO] cotuba-epub ........................ SUCCESS [  0.111 s]
```

```
[INFO] cotuba-html ......................... SUCCESS [  0.088 s]
[INFO] ------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------
[INFO] Total time: 2.273 s
```

Verifique os JARs gerados. Cada módulo terá, em seu diretório `target`, o seu respectivo JAR:

```
cotuba
│
├── pom.xml
│
├── cotuba-cli
│   ├── pom.xml
│   ├── src
│   └── target
│       ├── cotuba-cli-0.0.1-SNAPSHOT-distribution.zip
│       └── cotuba-cli-0.0.1-SNAPSHOT.jar
│
├── cotuba-core
│   ├── pom.xml
│   ├── src
│   └── target
│       └── cotuba-core-0.0.1-SNAPSHOT.jar
│
├── cotuba-pdf
│   ├── pom.xml
│   ├── src
│   └── target
│       └── cotuba-pdf-0.0.1-SNAPSHOT.jar
│
├── cotuba-epub
│   ├── pom.xml
│   ├── src
│   └── target
│       └── cotuba-epub-0.0.1-SNAPSHOT.jar
│
└── cotuba-html
   ├── pom.xml
   ├── src
```

```
    └── target
        └── cotuba-html-0.0.1-SNAPSHOT.jar
```

Há, porém, um problema: o arquivo `cotuba-cli-0.0.1-SNAPSHOT-distribution.zip` deveria conter todos os arquivos essenciais para a geração de ebooks com o Cotuba. Mas o ZIP que acabamos de gerar não contém os JARs de `cotuba-pdf`, `cotuba-epub` e `cotuba-html`, nem de suas respectivas dependências.

Para que o ZIP tenha todos os JARs necessários, devemos declarar os módulos de geração de ebooks como dependências no `pom.xml` de `cotuba-cli`:

```
<!-- cotuba/cotuba-cli/pom.xml -->

<dependency>
  <groupId>cotuba</groupId>
  <artifactId>cotuba-pdf</artifactId>
  <version>${project.version}</version>
</dependency>

<dependency>
  <groupId>cotuba</groupId>
  <artifactId>cotuba-epub</artifactId>
  <version>${project.version}</version>
</dependency>

<dependency>
  <groupId>cotuba</groupId>
  <artifactId>cotuba-html</artifactId>
  <version>${project.version}</version>
</dependency>
```

É importante notar que não há dependências de código de `cotuba-cli` diretamente aos módulos dos geradores de ebook. Ou seja, não há uso efetivo de classes que geram ebook no código relacionado à interface de linha de comando. A ligação entre a SPI e os *service providers* é feita pela classe `ServiceLoader`.

Tratam-se de dependências do Maven que, no momento do build, são usadas para gerar o ZIP do Cotuba, fazendo com que os JARs dos módulos geradores de ebook fiquem disponíveis no diretório `libs`.

Quando o build for refeito, o ZIP conterá os JARs dos módulos `cotuba-pdf`, `cotuba-epub` e `cotuba-html`.

10.7 O que aprendemos?

Neste capítulo, revelamos que os módulos da plataforma Java são os JARs e que podemos usar o mecanismo de modularização do Maven para criar um projeto modular. Além disso, abordamos os princípios de coesão e de acoplamento de módulos pregados por Uncle Bob.

No próximo capítulo, vamos pôr o design do nosso código à prova, criando uma interface Web para o Cotuba!

O código deste capítulo pode ser encontrado em: https://github.com/alexandreaquiles/solid-na-pratica/tree/cap10-modulos

Figura 10.12: Vídeo do capítulo

CAPÍTULO 11

Além da linha de comando: o Cotuba Web

Digamos que parte do time que desenvolve o Cotuba iniciou a implementação de uma UI Web, que é composta por 3 diferentes telas:

- Uma tela principal, que lista todos os livros cadastrados;

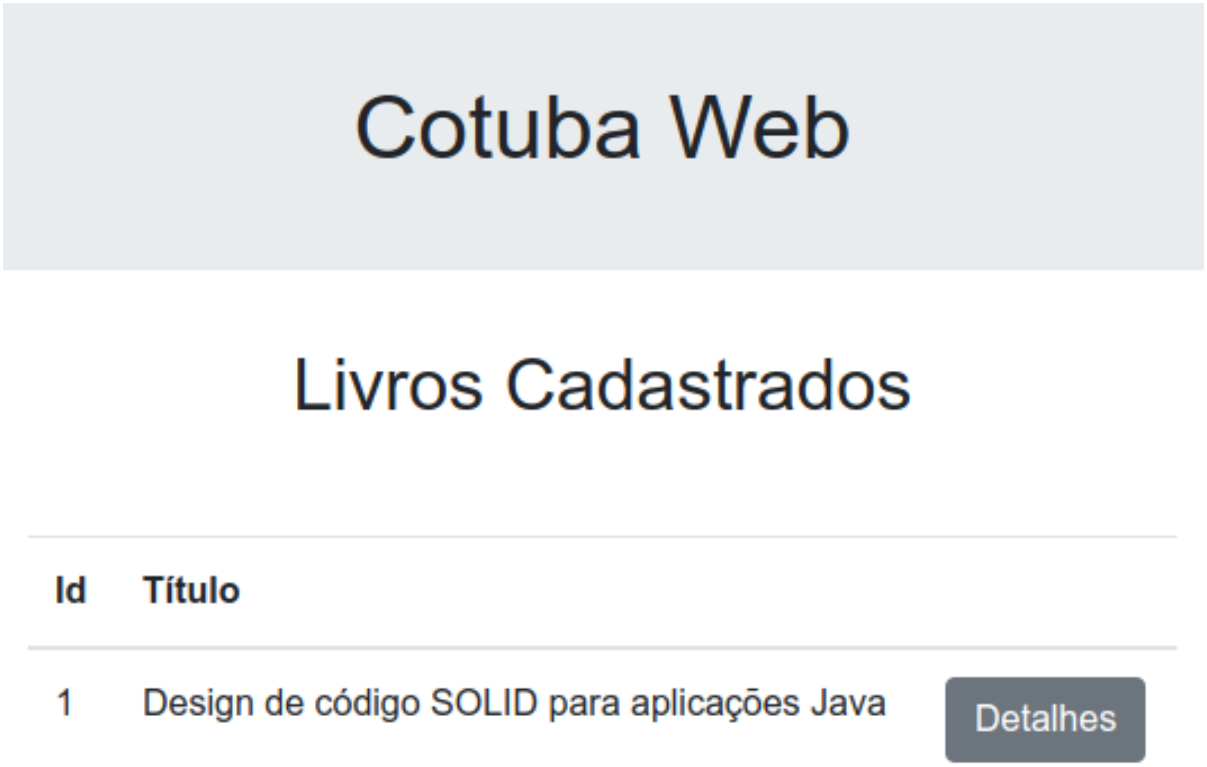

Figura 11.1: Livros cadastrados

- Uma tela que detalha os capítulos do livro e que permite gerar o EPUB e o PDF;

Figura 11.2: Detalhes do livro

- Uma tela de edição do título e Markdown de um capítulo.

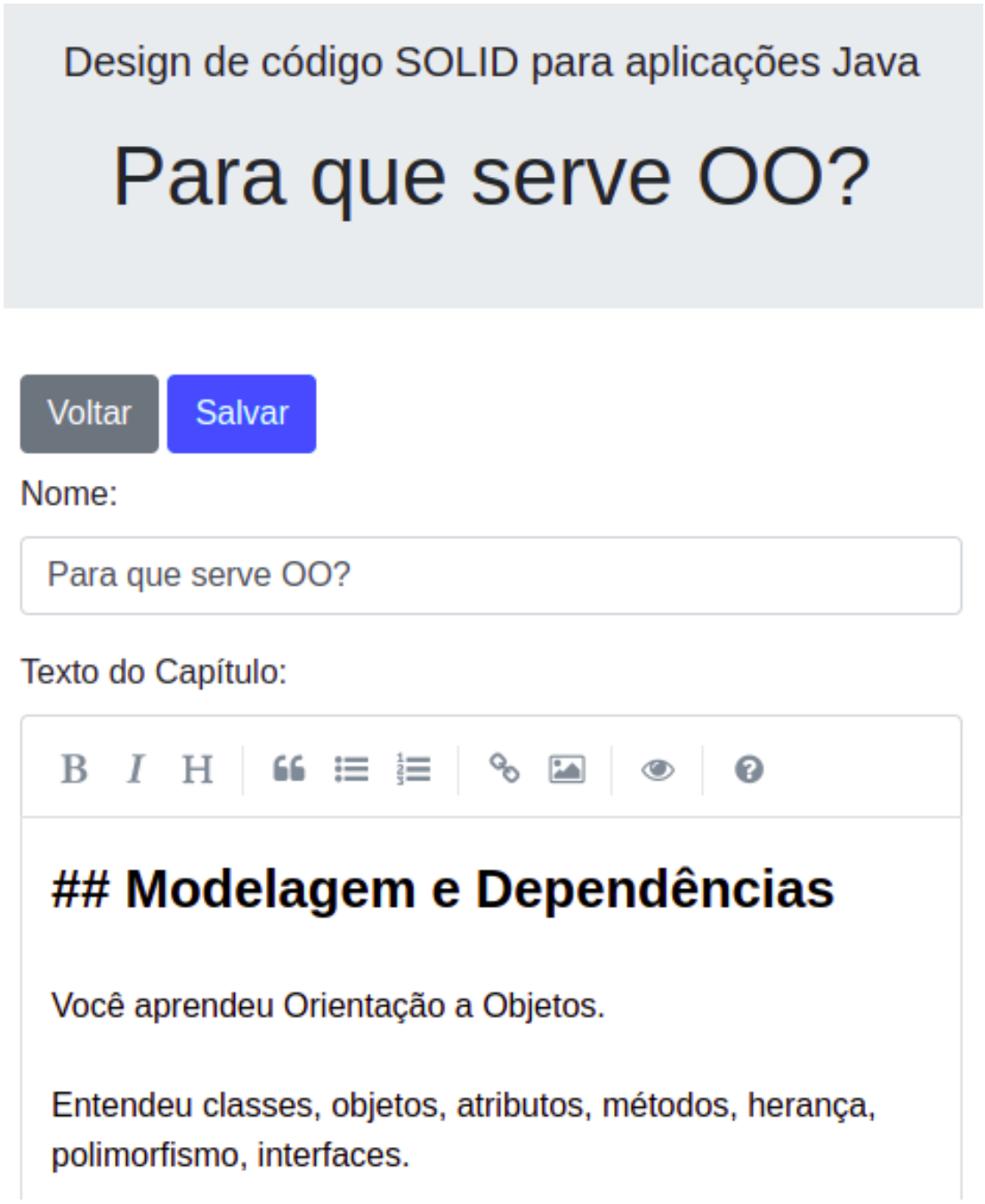

Figura 11.3: Detalhes de um capítulo

O projeto ainda está no início e, por isso, não há cadastro de novos livros, validações, nem uma série de outros detalhes. Algumas informações importantes sobre os requisitos da UI Web do Cotuba:

- apenas serão gerados ebooks nos formatos EPUB e PDF. O formato HTML não será suportado.
- na edição de capítulos, o conteúdo Markdown não contém um título (`#`), que deve ser prefixado logo antes de o Markdown ser renderizado em HTML.

A UI Web do Cotuba foi implementada no módulo `cotuba-web` . Teríamos, então, duas UIs:

- uma interface de linha de comando, implementada pelo módulo `cotuba-cli`
- uma interface Web, implementada pelo novo módulo `cotuba-web`

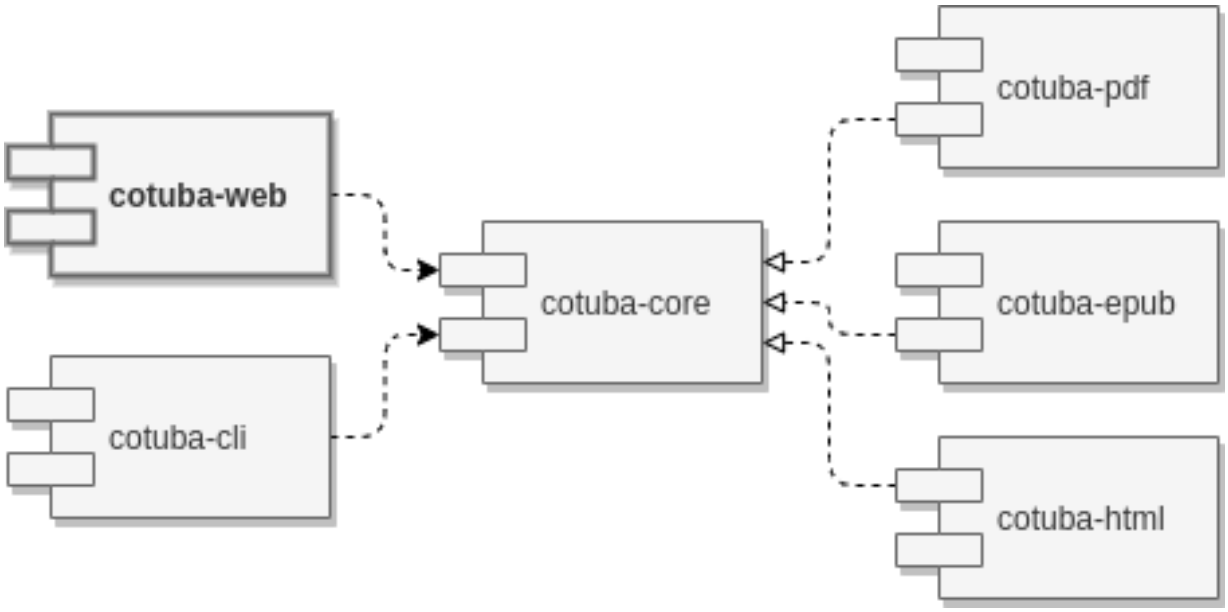

Figura 11.4: Um módulo para a UI Web

Obtendo o código da UI Web

O módulo `cotuba-web` pode ser obtido na branch `cotuba-web-inicial` do *projeto do Cotuba no GitHub* (https://github.com/alexandreaquiles/solid-na-pratica.git), que foi clonado nos capítulos iniciais. Nessa branch, também há o restante do código que fomos desenvolvendo no decorrer dos capítulos. Para obtê-lo, faça:

```
# cotuba

git checkout -f cotuba-web-inicial
```

A opção `-f` força a mudança de branch mesmo que haja mudanças ou arquivos não rastreados pelo Git. Em seguida, faça o build do supermódulo `cotuba`:

```
# cotuba

mvn clean install
```

Durante o build, o Maven tratará de baixar as dependências necessárias. A compilação deverá ser bem-sucedida:

```
[INFO] Reactor Summary for cotuba 0.0.1-SNAPSHOT:
[INFO]
[INFO] cotuba ............................. SUCCESS [  0.204 s]
[INFO] cotuba-core ........................ SUCCESS [  1.206 s]
[INFO] cotuba-pdf ......................... SUCCESS [  0.163 s]
[INFO] cotuba-epub ........................ SUCCESS [  0.101 s]
[INFO] cotuba-html ........................ SUCCESS [  0.105 s]
[INFO] cotuba-cli ......................... SUCCESS [  1.036 s]
[INFO] cotuba-web ......................... SUCCESS [  1.046 s]
[INFO] ------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------
[INFO] Total time:  4.072 s
```

O artefato gerado pelo `cotuba-web` será um *fat JAR* com cerca de 41 MB e que inclui tanto as classes da aplicação como a de todas as dependências e foi gerado pelo plugin Maven do Spring Boot. O local desse artefato será: `cotuba/cotuba-web/target/cotuba-web-0.0.1-SNAPSHOT.jar`.

Para executar a UI Web do Cotuba, podemos usar o seguinte comando:

```
# cotuba

java -jar cotuba-web/target/cotuba-web-0.0.1-SNAPSHOT.jar
```

Depois de várias mensagens, deve aparecer no seu Terminal algo como:

```
INFO 90398 --- [main] o.s.b.w.embedded.tomcat.TomcatWebServer  :
    Tomcat started on port(s): 8080 (http) with context path ''
INFO 90398 --- [main] cotuba.web.CotubaWebApplication  :
    Started Boot in 4.013 seconds (JVM running for 4.379)
```

Cotuba Web no ar! Não deixe de acessá-lo a partir da seguinte URL: http://localhost:8080/

Já há um livro cadastrado, o "Design de código SOLID para aplicações Java", que é inserido automaticamente pelos scripts de migração ao subirmos a aplicação pela primeira vez. Ainda não foi implementado o cadastro de novos livros.

Não deixe de abrir o projeto `cotuba` na sua IDE de preferência. Se desejar iniciar a UI Web do Cotuba pela IDE, execute a classe `CotubaWebApplication` do pacote `cotuba.web`!

Entendendo o código do Cotuba Web

A UI Web do Cotuba foi implementada usando as seguintes tecnologias:

- Spring Boot, com configurações em `src/main/resources/application.properties`;
- Thymeleaf para as telas, que ficam em `src/main/resources/templates`;
- Flyway para manter scripts SQL que criam e populam as tabelas e ficam em `src/main/resources/db/migration`;
- H2 como banco de dados em memória, cujos dados são limpos ao parar a aplicação;
- Spring Data JPA, para facilitar o código de acesso a dados.

O código do módulo `cotuba-web` está organizado no seguinte pacotes:

- `cotuba.web`, que contém a classe principal `CotubaWebApplication`, que inicializa o servidor Web;
- `cotuba.web.domain`, que contém as classes `Livro` e `Capitulo`, o pequeno modelo de domínio do Cotuba Web;
- `cotuba.web.repository`, que contém as interfaces `RepositorioDeCapitulos` e `RepositorioDeLivros`, que usam o Spring Data JPA para implementar o acesso a dados;
- `cotuba.web.controller`, que contém as classes `LivrosController` e `CapitulosController`, expõe as funcionalidades na Web, associando-as aos templates Thymeleaf;
- `cotuba.web.controller.form`, que contém a classe `CapituloForm`, que representa o formulário de cadastro de capítulos.

11.1 Gerando ebooks pela Web

Nosso objetivo é usar o Cotuba para gerar ebooks a partir dos livros cadastrados no Banco de Dados. Faremos a implementação passo a passo e poremos à prova a flexibilidade do design do Cotuba!

Implementando o download de ebooks

Vamos implementar um download para os ebooks que já geramos nos formatos PDF e EPUB. Não oferecemos suporte ao HTML de ebooks.

Para isso, vamos definir uma classe chamada `GeracaoDeLivrosController`, no pacote `cotuba.web.controller`. Vamos anotá-la com `@RestController`, indicando ao Spring que os dados devem ser serializados em vez de passados para um template Thymeleaf.

Definiremos URLs apropriadas para cada formato de ebook suportado. Por exemplo, para o livro "Design de código SOLID para aplicações Java", que já vem cadastrado no BD por meio dos scripts de migração do Flyway e cujo `id` é `1`, teríamos as seguintes URLs:

- `/livros/1/pdf` para fazer o download do PDF
- `/livros/1/epub` para fazer o download do EPUB

Por enquanto, vamos baixar um arquivo fixo para cada formato. No caso do PDF, o arquivo será chamado `book.pdf`. Já no caso do EPUB, será `book.epub`. Ambos os arquivos deverão estar no `Desktop` do usuário. Para obtermos o *home* do usuário, definiremos o valor da propriedade de sistema `user.home` em uma constante `USER_HOME`.

Obteremos os *bytes* dos arquivos por meio das classes utilitárias `Paths` e `Files` da API NIO do Java. Os métodos do controller

deverão retornar um `byte[]` e deverão ser anotados com `@GetMapping`, tanto para definir a URL e o `id` do livro como parâmetro como para definir o *media type* do retorno por meio da propriedade `produces`. Para definir o nome do arquivo a ser baixado pelo navegador, vamos definir o cabeçalho `Content-Disposition` por meio da `HttpServletResponse` injetada pelo Spring.

O código de `GeracaoDeLivrosController` será o seguinte:

```
// cotuba/cotuba-web
// cotuba.web.controller.GeracaoDeLivrosController

package cotuba.web.controller;

import org.springframework.http.HttpHeaders;
import org.springframework.web.bind.annotation.*;

import javax.servlet.http.HttpServletResponse;

import java.io.IOException;
import java.nio.file.*;

@RestController
public class GeracaoDeLivrosController {

  private static final String USER_HOME =
      System.getProperty("user.home");

  @GetMapping(value = "/livros/{id}/pdf",
      produces = "application/pdf")
  public byte[] geraPDF(@PathVariable("id") Long id,
      HttpServletResponse response) throws IOException {
    return ebookParaOFormato("pdf");
  }

  @GetMapping(value = "/livros/{id}/epub",
      produces = "application/epub+zip")
  public byte[] geraEPUB(@PathVariable("id") Long id,
      HttpServletResponse response) throws IOException {
    return ebookParaOFormato("epub");
```

```
  }

  private byte[] ebookParaOFormato(String formato,
      HttpServletResponse response) throws IOException {
    response.addHeader(HttpHeaders.CONTENT_DISPOSITION,
        "attachment;filename=book."+formato);
    Path ebook = Paths.get(USER_HOME, "Desktop/book."+formato);
    return Files.readAllBytes(ebook);
  }

}
```

Após reiniciarmos a aplicação pela IDE ou após novo build pela linha de comando, um clique nos botões de geração de PDF ou EPUB na página de detalhes do livro deve disparar um download do ebook!

Preparando o controller para chamar o Cotuba

Devemos obter os dados do livro e invocar a classe `Cotuba`, passando os parâmetros necessários. Primeiramente, vamos definir como dependência do módulo `cotuba-web` os módulos:

- `cotuba-core`, cujas classes serão chamadas diretamente pelo código Web;
- `cotuba-pdf` e `cotuba-epub`, que não serão chamados diretamente, mas cujos JARs precisam estar presentes para servirem como plugins.

```
<!-- cotuba/cotuba-web/pom.xml -->

<!-- inserido -->
<dependency>
  <groupId>cotuba</groupId>
  <artifactId>cotuba-core</artifactId>
  <version>${project.version}</version>
</dependency>

<!-- inserido -->
<dependency>
```

```
  <groupId>cotuba</groupId>
  <artifactId>cotuba-pdf</artifactId>
  <version>${project.version}</version>
</dependency>

<!-- inserido -->
<dependency>
  <groupId>cotuba</groupId>
  <artifactId>cotuba-epub</artifactId>
  <version>${project.version}</version>
</dependency>
```

Para chamar o Cotuba, vamos criar uma classe `GeracaoDeLivros`, em um novo pacote `cotuba.web.application`. Devemos anotá-la com `@Service`. Na nova classe, definiremos um método `geraLivro`, que recebe uma lista de `Capitulo` do pacote `cotuba.web.domain` e o formato do ebook. O retorno será um `Path` com o arquivo de saída:

```
// cotuba/cotuba-web
// cotuba.web.application.GeracaoDeLivros

package cotuba.web.application;

import cotuba.domain.FormatoEbook;
import cotuba.web.domain.Capitulo;
import java.nio.file.Path;
import java.util.List;
import org.springframework.stereotype.Service;

@Service
public class GeracaoDeLivros {

  public Path geraLivro(List<Capitulo> capitulos,
      FormatoEbook formato) {

    // chamada do Cotuba será feita aqui...

  }
}
```

O código não será compilado com sucesso ainda. Precisamos chamar o Cotuba e retornar o arquivo gerado para o ebook!

Na classe `GeracaoDeLivrosController`, devemos receber no construtor um `RepositorioDeLivros`, um `RepositorioDeCapitulos` e um `GeracaoDeLivros`, armazenando-os em atributos `final`. O Spring cuidará da injeção de dependências!

```
// cotuba/cotuba-web
// cotuba.web.controller.GeracaoDeLivrosController

import cotuba.web.application.GeracaoDeLivros; // inserido
import cotuba.web.repository.RepositorioDeCapitulos; // inserido
import cotuba.web.repository.RepositorioDeLivros; // inserido

@RestController
public class GeracaoDeLivrosController {

  private static final String USER_HOME =
      System.getProperty("user.home");

  // inserido
  private final RepositorioDeLivros repositorioDeLivros;
  private final RepositorioDeCapitulos repositorioDeCapitulos;
  private final GeracaoDeLivros geracao;

  // inserido
  public GeracaoDeLivrosController(
      RepositorioDeLivros repositorioDeLivros,
      RepositorioDeCapitulos repositorioDeCapitulos,
      GeracaoDeLivros geracao) {
    this.repositorioDeLivros = repositorioDeLivros;
    this.repositorioDeCapitulos = repositorioDeCapitulos;
    this.geracao = geracao;
  }

  // código omitido...

}
```

Os métodos de geração de PDF e EPUB do `GeracaoDeLivrosController` devem ser modificados, obtendo os capítulos a partir do `id` do livro recebido como parâmetro e, em seguida, invocando a geração de livros com o `FormatoEbook` adequado:

```
// cotuba/cotuba-web
// cotuba.web.controller.GeracaoDeLivrosController

// inserido
import cotuba.domain.FormatoEbook;
import cotuba.web.domain.Livro;
import org.springframework.http.HttpStatus;
import org.springframework.web.server.ResponseStatusException;

@RestController
public class GeracaoDeLivrosController {

  private static final String USER_HOME =
    System.getProperty("user.home");

  // código omitido...

  @GetMapping(value = "/livros/{id}/pdf",
      produces = "application/pdf")
  public byte[] geraPDF(@PathVariable("id") Long id,
      HttpServletResponse response) throws IOException {
    // modificado
    return ebookParaOFormato(FormatoEbook.PDF, response, id);
  }

  @GetMapping(value = "/livros/{id}/epub",
      produces = "application/epub+zip")
  public byte[] geraEPUB(@PathVariable("id") Long id,
      HttpServletResponse response) throws IOException {
    // modificado
    return ebookParaOFormato(FormatoEbook.EPUB, response, id);
  }

  // modificado
```

```
  private byte[] ebookParaOFormato(FormatoEbook formato,
      HttpServletResponse response, Long idLivro)
      throws IOException {
    response.addHeader(HttpHeaders.CONTENT_DISPOSITION,
                       "attachment;filename=book." +
                       formato.name().toLowerCase());

    Livro livro = repositorioDeLivros.findById(idLivro)
        .orElseThrow(() ->
            new ResponseStatusException(HttpStatus.NOT_FOUND));
    List<Capitulo> capitulos =
        repositorioDeCapitulos.findAllByLivroOrderByOrdem(livro);

    Path ebook = geracao.geraLivro(capitulos, formato);

    return Files.readAllBytes(ebook);
  }

}
```

Não é possível testar ainda, pois há um erro de compilação em `GeracaoDeLivros`.

Chamando o Cotuba a partir da Web

Como chamar o Cotuba em `GeracaoDeLivros`? A primeira coisa é instanciá-lo no método `geraLivro`. Em seguida, precisamos invocar o método `executa` de `Cotuba`, que recebe como parâmetro uma implementação da interface `ParametrosCotuba`:

```
// cotuba/cotuba-web
// cotuba.web.application.GeracaoDeLivros@Service

import cotuba.application.Cotuba; // inserido
import cotuba.application.ParametrosCotuba; // inserido

@Service
public class GeracaoDeLivros {

  public Path geraLivro(List<Capitulo> capitulos,
```

```
      FormatoEbook formato) {

    // inserido
    var cotuba = new Cotuba();
    ParametrosCotuba parametros = // o que colocar aqui???
    cotuba.executa(parametros);

 }

}
```

Precisamos criar uma implementação da interface `ParametrosCotuba`. Vamos defini-la no pacote `cotuba.web.application` e chamá-la de `ParametrosCotubaWeb`:

```
// cotuba/cotuba-web
// cotuba.web.application.ParametrosCotubaWeb

package cotuba.web.application;

import java.nio.file.Path;

import cotuba.domain.FormatoEbook;
import cotuba.application.ParametrosCotuba;

public class ParametrosCotubaWeb implements ParametrosCotuba {

  @Override
  public FormatoEbook getFormato() {
  }

  @Override
  public Path getArquivoDeSaida() {
  }

  @Override
  public Path getDiretorioDosMD() {
  }

}
```

Na classe `ParametrosCotubaWeb`, devemos receber o `FormatoEbook` como parâmetro do construtor, definindo-o em um atributo `final`. O `formato` recebido no construtor deve ser retornado no método `getFormato`:

```
// cotuba/cotuba-web
// cotuba.web.application.ParametrosCotubaWeb

public class ParametrosCotubaWeb implements ParametrosCotuba {

  private final FormatoEbook formato; // inserido

  // inserido
  public ParametrosCotubaWeb(FormatoEbook formato) {
    this.formato = formato;
  }

  @Override
  public FormatoEbook getFormato() {
    return formato; // modificado
  }

  // código omitido...

}
```

Para definir o arquivo de saída, vamos criar um método auxiliar `criaArquivoTemporario`, que cria um arquivo em um diretório temporário cujo prefixo é `ebooks` usando o método `createTempDirectory` da classe `Files` da API `java.nio`. O nome do arquivo deve ser `book.pdf` para a geração de PDFs e `book.epub` para EPUBs.

No construtor de `ParametrosCotubaWeb`, vamos invocar o novo método, armazenando o `Path` em um atributo `arquivoDeSaida`, retornando esse atributo no método `getArquivoDeSaida`:

```
// cotuba/cotuba-web
// cotuba.web.application.ParametrosCotubaWeb
```

```java
import java.io.IOException; // inserido
import java.nio.file.Files; // inserido

public class ParametrosCotubaWeb implements ParametrosCotuba {

  private final FormatoEbook formato;
  private final Path arquivoDeSaida; // inserido

  public ParametrosCotubaWeb(FormatoEbook formato) {
    this.formato = formato;
    this.arquivoDeSaida = criaArquivoTemporario(); // inserido
  }

  // código omitido...

  @Override
  public Path getArquivoDeSaida() {
    return arquivoDeSaida; // modificado
  }

  // inserido
  private Path criaArquivoTemporario() {
    try {
      Path diretorioTemporario =
          Files.createTempDirectory("ebooks");
      String nomeDoArquivoDeSaida =
          "book." + formato.name().toLowerCase();
      return diretorioTemporario.resolve(nomeDoArquivoDeSaida);
    } catch (IOException ex) {
        throw new IllegalStateException(ex);
    }
  }

}
```

Os arquivos temporários de saída seriam semelhantes aos seguintes:

- para um PDF, algo como `/tmp/ebooks1653289186159041686/book.pdf`

- para um EPUB, algo como
  `/tmp/ebooks437987585657386531 9/book.epub`

Ainda é preciso definir o diretório onde estão os arquivos `.md` . Mas peraí...

11.2 Quando nosso design não antecipa as mudanças

Precisamos definir uma implementação para o método `getDiretorioDosMD` na classe `ParametrosCotubaWeb`. Mas onde estão os arquivos Markdown do Cotuba Web? No Banco de Dados! Então, temos dois clientes que invocam o Cotuba: a interface de linha de comando e a interface Web. Cada cliente obtém os arquivos MD de maneiras distintas:

- o `cotuba-cli` obtém os `.md` de um diretório
- o `cotuba-web` obtém os `.md` da tabela de capítulos de seu Banco de Dados

Um ideia de implementação para `getDiretorioDosMD` é criar, a partir dos capítulos do Banco de Dados, arquivos `.md`, salvando-os em um diretório. Mas será que é a melhor opção? Será que não há um problema no design? Há algum um **ponto de mudança que não foi antecipado**?

A classe que usa o diretório com os arquivos `.md` é a `RenderizadorMDParaHTML`. Se repararmos bem, essa classe quebra um dos princípios do SOLID. Deixamos essa dica no capítulo sobre SRP. Há dois motivos para mudá-la:

- uma mudança na maneira de renderizar os arquivos MD para HTML, o que parece alinhado com o objetivo da classe;
- uma mudança na maneira de obter esses arquivos MD, que parece algo fora das responsabilidades dessa classe.

Há responsabilidades demais na classe `RenderizadorMDParaHTML`: uma violação do SRP! E uma violação do DIP, já que lidar com arquivos é código de baixo nível.

Mudanças inesperadas e design incremental

Às vezes é **difícil antecipar mudanças** que acontecerão no problema que estamos resolvendo e o respectivo impacto no código. Consequentemente, não é trivial criar um design que tem flexibilidade onde realmente é necessário e que "abraça a mudança".

Quando detectarmos um problema no design, em vez de tentarmos encaixar o código no design já existente, devemos corrigi-lo. Assim, com o tempo, teremos flexibilidade nos pontos certos.

> *Escrever código de qualidade é sempre incremental; você modela, observa seu modelo, aprende com ele e o melhora.*
>
> Maurício Aniche, no livro *OO e SOLID para Ninjas* (ANICHE, 2015)

Mais abstração, mais flexibilidade

Para trazer flexibilidade na obtenção dos arquivos MD, vamos criar uma abstração chamada `RepositorioDeMDs` no pacote `cotuba.application` do módulo `cotuba-core`. Nessa interface, definiremos um método `obtemMDsDosCapitulos`, que deve retornar uma lista com o conteúdo Markdown dos capítulos.

```
// cotuba/cotuba-core
// cotuba.application.RepositorioDeMDs

package cotuba.application;

import java.util.List;

public interface RepositorioDeMDs {

  List<String> obtemMDsDosCapitulos();

}
```

Podemos trocar o diretório dos `ParametrosCotuba` pela nova abstração de a obtenção de MDs:

```
// cotuba/cotuba-core
// cotuba.application.ParametrosCotuba

public interface ParametrosCotuba {

  Path getDiretorioDosMD();
  RepositorioDeMDs getRepositorioDeMDs();

  FormatoEbook getFormato();

  Path getArquivoDeSaida();

}
```

A classe `RenderizadorMDParaHTML` deve ser atualizada para receber um `RepositorioDeMDs` em seu método `renderiza`. A nova abstração deve ser usada no lugar do método auxiliar `obterArquivosMD`. Além disso, não precisaremos passar o `arquivoMD` para o método auxiliar `renderizaParaHTML`:

```
// cotuba/cotuba-core
// cotuba.md.RenderizadorMDParaHTML

import cotuba.application.RepositorioDeMDs; // inserido

public class RenderizadorMDParaHTML {

  public List<Capitulo> renderiza(Path diretorioDosMD) {
  public List<Capitulo> renderiza(
        RepositorioDeMDs repositorioDeMDs) {
    return repositorioDeMDs.obtemMDsDosCapitulos().stream()
        .map(conteudoMD -> {
          CapituloBuilder capituloBuilder
              = new CapituloBuilder();
          Node document = parseDoMD(conteudoMD, capituloBuilder);
          renderizaParaHTML(capituloBuilder, document);
          return capituloBuilder.constroi();
```

```
        }).toList();
  }

}
```

O método auxiliar `obterArquivosMD`, que lê os arquivos `.md` de um diretório, deve ser removido de `RenderizadorMDParaHTML`. Essa implementação ficará em outra classe!

O método auxiliar `parseDoMD` deve receber uma `String` com o conteúdo Markdown, em vez de um `Path`. Deve ser usado o método apropriado do `Parser` da biblioteca CommonMark. Além disso, o tratamento de exceções pode ser simplificado:

```
// cotuba/cotuba-core
// cotuba.md.RenderizadorMDParaHTML

public class RenderizadorMDParaHTML {

  // código omitido...

  private Node parseDoMD(Path arquivoMD, CapituloBuilder capituloBuilder) {
  private Node parseDoMD(String conteudoMD,
      CapituloBuilder capituloBuilder) {

    // código omitido...
    try {

      document = parser.parseReader(Files.newBufferedReader(arquivoMD));
      document = parser.parse(conteudoMD);

      // código omitido...
    } catch (Exception ex) {
      throw new IllegalStateException("Erro ao fazer parse do arquivo " + arquivoMD, ex);
      throw new IllegalStateException(
          "Erro ao fazer parse de MD", ex);
    }
  }
```

```
  // código omitido...

}
```

Ainda na classe `RenderizadorMDParaHTML`, a assinatura e o tratamento de exceções do método auxiliar `renderizaParaHTML` podem ser simplificados:

```
// cotuba/cotuba-core
// cotuba.md.RenderizadorMDParaHTML

public class RenderizadorMDParaHTML {

  // código omitido...

  private void renderizaParaHTML(Path arquivoMD,
      CapituloBuilder capituloBuilder, Node document) {
    try {
      // código não modificado...
    } catch (Exception ex) {
      throw new IllegalStateException("Erro ao renderizar para HTML o arquivo " + arquivoMD, ex);
      throw new IllegalStateException(
        "Erro ao renderizar MD para HTML", ex);
    }
  }
}
```

Os imports desnecessários, de classes dos pacotes `java.io` e `java.nio.file`, devem ser removidos de `RenderizadorMDParaHTML`. Em seguida, devemos modificar o método `executa` da classe `Cotuba` para obter uma implementação de `RepositorioDeMDs` dos `ParametrosCotuba` e repassá-la para o `RenderizadorMDParaHTML`:

```
// cotuba/cotuba-core
// cotuba.application.Cotuba

public class Cotuba {

  public void executa(ParametrosCotuba parametros) {
```

```
    FormatoEbook formato = parametros.getFormato();

    Path diretorioDosMD = parametros.getDiretorioDosMD();
    RepositorioDeMDs repositorioDeMDs =
        parametros.getRepositorioDeMDs();

    Path arquivoDeSaida = parametros.getArquivoDeSaida();

    var renderizador = new RenderizadorMDParaHTML();

    List<Capitulo> capitulos = renderizador.renderiza(diretorioDosMD);
    List<Capitulo> capitulos =
        renderizador.renderiza(repositorioDeMDs);

    // código omitido...

  }

}
```

O código do módulo `cotuba-core` deve ser compilado com sucesso. Já os módulos `cotuba-cli` e `cotuba-web` devem apresentar erros de compilação.

Ajustando a linha de comando

Precisamos ajustar o módulo `cotuba-cli` para usar a nova abstração de repositórios de arquivos Markdown.

No pacote `cotuba.cli` do módulo `cotuba-cli`, vamos criar a classe `MDsDoDiretorio` que implementa a interface `RepositorioDeMDs`. No construtor, a nova classe deve receber um `Path` com o diretório onde estão os MDs, que deve ser armazenado em um atributo `final`. O método `obtemMDsDosCapitulos` deve retornar uma lista com o conteúdo de cada arquivo do diretório:

```
// cotuba/cotuba-cli
// cotuba.cli.MDsDoDiretorio
package cotuba.cli;
```

```java
import cotuba.application.RepositorioDeMDs;
import java.io.IOException;
import java.nio.file.*;
import java.util.List;
import java.util.stream.Stream;

class MDsDoDiretorio implements RepositorioDeMDs {

  private final Path diretorioDosMD;

  public MDsDoDiretorio(Path diretorioDosMD) {
    this.diretorioDosMD = diretorioDosMD;
  }

  @Override
  public List<String> obtemMDsDosCapitulos() {

    PathMatcher matcher =
        FileSystems.getDefault().getPathMatcher("glob:**/*.md");
    try (Stream<Path> arquivosMD = Files.list(diretorioDosMD)) {
      return arquivosMD
          .filter(matcher::matches)
          .sorted()
          .map(arquivoMD -> {
            try {
              return new String(Files.readAllBytes(arquivoMD));
            } catch (Exception ex) {
              throw new IllegalStateException(
                    "Erro ao ler arquivo " + arquivoMD, ex);
            }
          })
          .toList();
    } catch (IOException ex) {
      throw new IllegalStateException(
            "Erro tentando encontrar arquivos .md em " +
                diretorioDosMD.toAbsolutePath(), ex);
    }

  }
}
```

Na classe `LeitorOpcoesCLI`, devemos implementar o método `getRepositorioDeMDs` definido no novo contrato de `ParametrosCotuba` e retornar uma instância de `MDsDoDiretorio`:

```
// cotuba/cotuba-cli
// cotuba.cli.LeitorOpcoesCLI

import cotuba.application.RepositorioDeMDs;

class LeitorOpcoesCLI implements ParametrosCotuba {

  // código omitido...

  @Override
  public RepositorioDeMDs getRepositorioDeMDs() {
    return new MDsDoDiretorio(diretorioDosMD);
  }

  // código omitido...

}
```

Pronto! A geração de PDFs e EPUBs pela linha de comando deve funcionar. Agora podemos partir para a Web!

Gerando ebooks a partir do Cotuba Web

Continuando a implementação da geração de ebooks a partir da UI Web do Cotuba, vamos definir uma implementação de `RepositorioDeMDs`, que obtém o conteúdo Markdown a partir do Banco de Dados.

Para isso, vamos definir uma nova classe `MDsDoBancoDeDados` no pacote `cotuba.web.application`, com um construtor que recebe uma `List` de `Capitulo` do pacote `cotuba.web.domain`.

Essa nova classe deve implementar a interface `RepositorioDeMDs` e, no método `obtemMDsDosCapitulos`, deve montar uma lista com os conteúdos Markdown dos capítulos do livro. É importante lembrar

que os capítulos não contêm um título (`#`), que deve ser prefixado antes de ser adicionado à lista de retorno:

```
// cotuba/cotuba-web
// cotuba.web.application.MDsDoBancoDeDados

package cotuba.web.application;

import cotuba.application.RepositorioDeMDs;
import cotuba.web.domain.Capitulo;
import org.springframework.stereotype.Service;

import java.util.List;

@Service
public class MDsDoBancoDeDados implements RepositorioDeMDs {

  private final List<Capitulo> capitulos;

  public MDsDoBancoDeDados(List<Capitulo> capitulos) {
    this.capitulos = capitulos;
  }

  @Override
  public List<String> obtemMDsDosCapitulos() {
    return capitulos.stream()
        .map(capitulo -> "# " + capitulo.getNome() + "\n"
            + capitulo.getMarkdown())
        .toList();
  }

}
```

Vamos alterar a classe `ParametrosCotubaWeb` , recebendo uma instância de `MDsDoBancoDeDados` pelo construtor e retornando-a no método `getRepositorioDeMDs` , que está definido no contrato da interface `ParametrosCotuba` :

```
public class ParametrosCotubaWeb implements ParametrosCotuba {

  private final FormatoEbook formato;
```

```
  private final Path arquivoDeSaida;
  private final MDsDoBancoDeDados mDsDoBancoDeDados; // inserido

  public ParametrosCotubaWeb(FormatoEbook formato,
      MDsDoBancoDeDados mDsDoBancoDeDados) { // modificado
    this.formato = formato;
    this.arquivoDeSaida = criaArquivoTemporario();
    this.mDsDoBancoDeDados = mDsDoBancoDeDados; // inserido
  }

  // código omitido...

  @Override
  public Path getDiretorioDosMD() {
  }

  // inserido
  @Override
  public RepositorioDeMDs getRepositorioDeMDs() {
    return mDsDoBancoDeDados;
  }

}
```

No método `geraLivro` da classe `GeracaoDeLivros`, devemos instanciar um `MDsDoBancoDeDados` usando os capítulos recebidos como parâmetro. Em seguida, devemos criar uma instância de `ParametrosCotubaWeb`, passando-a para o método `executa` de `Cotuba`. Ao final, devemos retornar o `Path` do ebook gerado:

```
// cotuba/cotuba-web
// cotuba.web.application.GeracaoDeLivros

@Service
public class GeracaoDeLivros {

  // código omitido...

  public Path geraLivro(List<Capitulo> capitulos,
      FormatoEbook formato) {
```

```
    MDsDoBancoDeDados mDsDoBancoDeDados =
        new MDsDoBancoDeDados(capitulos); // inserido

    var cotuba = new Cotuba();

    // modificado
    ParametrosCotuba parametros =
        new ParametrosCotubaWeb(formato, mDsDoBancoDeDados);

    cotuba.executa(parametros);

    return parametros.getArquivoDeSaida(); // inserido
  }

}
```

Com o build feito e o servidor Web rodando, podemos testar a geração do PDF e do EPUB pela Web usando os botões apropriados da seguinte URL: http://localhost:8080/livros/1

Pronto! Agora o Cotuba foi adaptado para que a geração de ebooks seja feita a partir da Web!

11.3 O que aprendemos?

Neste capítulo, começamos executando uma UI Web para o Cotuba. Em seguida, implementamos a geração de ebooks a partir da Web. Mas não sem esforço: descobrimos violações do SRP e do DIP no renderizador de Markdown para HTML, que obtinha o conteúdo Markdown somente de arquivos. Criamos uma nova abstração para um repositório de Markdowns, tornando o código mais flexível. Adaptamos a UI de linha de comando para implementar a abstração por meio da leitura de arquivos. E, finalmente, fizemos com que a UI Web obtivesse os Markdowns do Banco de Dados. Ufa!

A lição é que, mesmo nos esforçando para seguir bons princípios e deixar nosso código flexível e adaptável, nem sempre conseguimos antecipar o ponto exato em que ocorrerão mudanças no nosso código.

No próximo capítulo, estudaremos quais os problemas do Classpath e pensaremos no encapsulamento de módulos. Até mais!

O código deste capítulo pode ser encontrado em: https://github.com/alexandreaquiles/solid-na-pratica/tree/cap11-cotuba-web

Figura 11.5: Vídeo do capítulo

CAPÍTULO 12

Módulos com o Java Platform Module System (JPMS)

Os módulos da plataforma Java são os JARs, conforme estudamos em um capítulo anterior. E o que há dentro de JARs? JARs contêm arquivos com a extensão `.class` e outros recursos como arquivos de configuração e imagens.

Os arquivos `.class`, por sua vez, contêm os **tipos** do Java, que podem ser: classes, interfaces, enums, anotações e, a partir do Java 16, records. Arquivos `.class` são definidos em diretórios: os **pacotes**. Portanto, podemos afirmar que pacotes contêm tipos.

12.1 Encapsulamento de módulos e o Classpath

Os modificadores de acesso em um código Java ajudam a promover o encapsulamento. E quais são os modificadores de acesso de um tipo? Um tipo tem apenas dois níveis de modificadores de acesso:

- o padrão, também conhecido como *package private*, que torna o tipo acessível apenas no pacote em que está definido;
- `public`, que torna o tipo acessível a tipos de qualquer outro pacote.

Uma pergunta: em projetos de que você participou, já viu alguma classe ser definida sem o modificador `public`?

É muito raro! Tipos, como classes, raramente são definidos com o modificador de acesso padrão, *package private*. Uma classe criada por uma IDE, por exemplo, já é automaticamente definida como pública, em geral.

Em um projeto modularizado, **termos apenas tipo públicos causa problemas**: uma parcela dos tipos deveriam ser detalhes internos de um módulo mas, se todos são públicos, podem ser acessados por qualquer outro módulo. Com isso, é grande a tentação de usá-los de maneiras indevidas em outros módulos.

Até poderíamos colocar todos os tipos de um módulo em um mesmo pacote. Assim, poderíamos tornar públicos apenas os tipos que poderiam ser usados por outros módulos. Mas, convenhamos, deixar tudo no mesmo pacote deixa o código desorganizado e difícil de entender, não é mesmo?

Explorando a falta de encapsulamento dos módulos

O módulo Maven responsável pela UI de linha de comando, o `cotuba-cli`, depende de `cotuba-core` e usa tipos como a classe `Cotuba`, a enum `FormatoEbook` e a interface `ParametrosCotuba`. Até aí tudo bem!

Mas podemos acessar, em `cotuba-cli`, detalhes internos do módulo Maven `cotuba-core` como `CapituloBuilder` e `RenderizadorMDParaHTML`:

```
// cotuba-cli
// cotuba.cli.Main

// código omitido...

import cotuba.domain.builder.CapituloBuilder; // inserido
import cotuba.md.RenderizadorMDParaHTML; // inserido

public class Main {

  public static void main(String[] args) {

    CapituloBuilder builder = new CapituloBuilder(); // opa!
    System.out.println(builder);

    RenderizadorMDParaHTML md =
        new RenderizadorMDParaHTML(); // eita!
    System.out.println(md);

    // código omitido...

  }

}
```

O código anterior seria tanto compilado como executado sucesso. Não é feito nada útil, mas teríamos na saída padrão o resultado do `toString` de objetos das classes `CapituloBuilder` e `RenderizadorMDParaHTML`:

```
cotuba.domain.builder.CapituloBuilder@6e5e91e4
cotuba.md.RenderizadorMDParaHTML@2cdf8d8a
```

As dependências declaradas pelo Maven influenciam na compilação. Por exemplo, o módulo Maven `cotuba-core` não poderia usar classes de `cotuba-cli`, apenas o contrário. Mas como as classes `CapituloBuilder` e `RenderizadorMDParaHTML` são públicas, podem ser usadas pelo módulo Maven `cotuba-cli` ou qualquer outro módulo que tenha como dependência `cotuba-core` em seu `pom.xml`.

Detalhes que deveriam estar escondidos podem ser acessados e isso não é bom para a manutenibilidade do nosso código. Há uma **quebra do encapsulamento**.

Acessando dependências transitivas

É importante notar que as dependências transitivas, as dependências das dependências, também podem ser acessadas.

Por exemplo, o módulo Maven `cotuba-cli` tem a biblioteca CommonMark como dependência transitiva, já que é uma dependência de `cotuba-core`. Portanto, poderíamos utilizar o `Parser` ou o `HtmlRenderer` do CommonMark no módulo de linha de comando:

```
// cotuba-cli
// cotuba.cli.Main

// código omitido...

import org.commonmark.parser.Parser; // inserido
import org.commonmark.renderer.html.HtmlRenderer; // inserido

public class Main {

  public static void main(String[] args) {

    // código omitido...

    Parser parser = new Parser.Builder().build(); // nossa!
    System.out.println(parser);

    HtmlRenderer htmlRenderer =
```

```
            new HtmlRenderer.Builder().build(); // vixe!
        System.out.println(htmlRenderer);

        // código omitido...

    }

}
```

Depois de compilado o código anterior, ao executá-lo, seria impresso o resultado da invocação do `toString` para objetos das classes `Parser` e `HtmlRenderer`:

```
org.commonmark.parser.Parser@30946e09
org.commonmark.renderer.html.HtmlRenderer@5cb0d902
```

O código anterior é compilado e executado com sucesso, mas a biblioteca CommonMark não faz parte das dependências declaradas diretamente no `pom.xml` do módulo Maven `cotuba-cli`. Porém, faz parte de suas dependências transitivas.

Quem é o culpado por essa falta de encapsulamento?

O problema do Classpath

O Classpath é a lista de módulos (JARs) que podem ser utilizados na compilação e na execução. É definido pela opção `-cp` dos comando `javac` e `java`, que indica os JARs e diretórios onde podem ser encontrados arquivos `.class` que serão disponibilizados. Ferramentas como IDEs e o Maven também usam o Classpath, mas o definem de maneira oculta.

No script de execução do módulo Maven `cotuba-cli`, usamos o Classpath para indicar a pasta `libs` como fonte dos JARs usados na aplicação:

```
# cotuba/cotuba-cli/src/scripts/cotuba.sh

java -cp "libs/*" cotuba.Main "$@"
```

Com o Classpath, classes e outros tipos com o modificador de acesso `public` de um JAR serão acessíveis a tipos de quaisquer outros JARs. O Classpath fornece simplesmente uma lista de todos os tipos definidos nos JARs encontrados. Não há nada que represente o JAR de origem de um tipo nem as dependências entre os JARs. Não é possível criar uma funcionalidade acessível dentro de todos os pacotes de um JAR, mas não fora dele. Isso enfraquece o encapsulamento em uma aplicação Java.

Nicolai Parlog, no livro *The Java Module System* (PARLOG, 2018), elenca os seguintes problemas no Classpath:

- *Encapsulamento fraco entre JARs*: conforme estudamos, o Classpath é uma grande lista de tipos (classes, interfaces etc.). Os tipos públicos são visíveis por quaisquer outros tipos de qualquer outro JAR.
- *Ausência de representação das dependências entre JARs*: não há como declarar de quais outros JARs um determinado JAR depende apenas com o Classpath.
- *Ausência de checagens automáticas de segurança*: o encapsulamento fraco dos JARs permite que código malicioso acesse e manipule funcionalidade crítica. Porém, é possível implementar manualmente checagens de segurança.
- *Sombreamento de tipos com o mesmo nome*: no caso de JARs que definem duas classes, ou outros tipos, com o mesmo nome, apenas uma delas é tornada disponível. E não é possível saber qual.
- *Conflitos entre versões diferentes do mesmo JAR*: duas versões do mesmo JAR no Classpath levam a comportamentos imprevisíveis.
- *JRE é rígida*: não é possível disponibilizar no Classpath um subconjunto das bibliotecas padrões do Java. Muitos tipos não utilizados ficam acessíveis.
- *Performance ruim no startup*: apesar de os *class loaders* serem *lazy*, carregando tipos só no seu primeiro uso, muitos já são carregadas ao iniciar uma aplicação.

- *Class loading complexo*: é possível criar uma hierarquia de *class loaders*, algo comumente feito por servidores de aplicação. Essa hierarquia pode resultar em comportamentos difíceis de entender e em erros inesperados.

Quebrando o encapsulamento radicalmente com Reflection

Conseguimos compilar com sucesso código que usa qualquer tipo público das dependências, diretas ou transitivas, declaradas no `pom.xml` de um módulo Maven. Isso acontece porque o `maven-compiler-plugin`, responsável por definir o Classpath durante a compilação, configura todos os JARs das dependências diretas e transitivas.

No caso de não existir **nenhuma referência** a um JAR nem nas dependências diretas, nem nas transitivas, o código **não será compilado**.

Considere o módulo Maven `cotuba-pdf`, que tem como dependências apenas `cotuba-core` e a biblioteca iText pdfHtml. Não é possível usar diretamente no código classes que **não** estão entre suas dependências diretas nem transitivas. Por exemplo, se tentarmos instanciar as classes `Main` de `cotuba-cli` e `Options` de Apache Commons CLI, teríamos problemas ao compilar:

```
// cotuba-pdf
// cotuba.pdf.GeradorPDF

import cotuba.cli.Main; // error: package does not exist
import org.apache.commons.cli.Options; // error: package does not exist

public class GeradorPDF implements GeradorEbook {

  @Override
  public void gera(Ebook ebook) {

    Main main = new Main(); // error: cannot find symbol
    System.out.println(main);
```

```
    Options options = new Options(); // error: cannot find symbol
    System.out.println(options);

    // código omitido...

  }

}
```

Não temos as classes `Main` e `Options` disponíveis no Classpath configurado pelo Maven durante a compilação de `cotuba-pdf`. Por isso, são apresentados os erros de compilação.

Porém, se os JARs que contêm as classes `Main` e `Options` estiverem disponíveis no Classpath durante o *runtime*, é possível manipulá-las usando a Reflection API do Java:

```
// cotuba-pdf
// cotuba.pdf.GeradorPDF

import java.lang.reflect.InvocationTargetException; // inserido

public class GeradorPDF implements GeradorEbook {

  @Override
  public void gera(Ebook ebook) {

    try {

      // puxa!
      Class mainClass = Class.forName("cotuba.cli.Main");
      Object main =
          mainClass.getDeclaredConstructor().newInstance();
      System.out.println(main);

      // caramba!
      Class optionsClass =
          Class.forName("org.apache.commons.cli.Options");
      Object options =
          optionsClass.getDeclaredConstructor().newInstance();
      System.out.println(options);
```

```
    } catch (ClassNotFoundException | InstantiationException|
              IllegalAccessException | NoSuchMethodException |
              InvocationTargetException ex) {
      throw new IllegalStateException(ex);
    }

    // código omitido...

  }

}
```

O código anterior é compilado com sucesso e, ao ser executado, é adicionado na saída padrão algo como:

```
cotuba.cli.Main@20398b7c
[ Options: [ short {} ] [ long {} ]
```

São os resultados da chamada do `toString` para objetos das classes `Main` e `Options`!

> Se tentarmos executar o código anterior de `GeradorPDF` a partir da UI Web do Cotuba, ocorrerá uma `ClassNotFoundException`, já que nem a classe `Main` nem a classe `Options` estão entre as dependências do módulo Maven `cotuba-web`.

Mas como é possível o trecho de código anterior ser compilado sem erros? É que não existem referências diretas às classes do módulo Maven `cotuba-cli`, nem de Apache Commons CLI. É somente o *fully qualified name* dessas classes que é usado como parâmetros de classes da Reflection API.

Há a possibilidade de usar a Reflection API para manipular qualquer classe disponível no Classpath. Isso acontece porque o Classpath é somente uma lista de classes, sem qualquer referência a seu JAR de origem. O Classpath não guarda a informação que as classes `Main` e `Options` vieram do `cotuba-cli-0.0.1-SNAPSHOT.jar` e `commons-cli-`

`1.4.jar` , respectivamente. Há uma **quebra do encapsulamento no runtime**!

12.2 JPMS, um sistema de módulos para o Java

A partir do Java 9, o Java inclui um sistema de módulos bastante poderoso: o Java Platform Module System (JPMS). Um módulo JPMS é um JAR que define:

- um nome único para o módulo;
- dependências a outros módulos;
- pacotes exportados, cujos tipos públicos são acessíveis por outros módulos.

O nome de um módulo JPMS só pode conter caracteres alfanuméricos e pontos. Não pode conter hífens, nem nenhum outro caracter especial. Portanto:

- `casadocodigo-core` **não** é um nome válido para um módulo JPMS;
- `casadocodigo.core` é um nome válido para um módulo JPMS.

Considere um módulo JPMS com o nome `casadocodigo.core`, que exporta o pacote `br.com.casadocodigo.domain` e requer o módulo JPMS `casadocodigo.persistencia`. Para definir essas informações, o código-fonte desse módulo JPMS deve definir um arquivo `module-info.java`, com a seguinte estrutura:

```
module casadocodigo.core {

  exports br.com.casadocodigo.domain;

  requires casadocodigo.persistencia;

}
```

Após a compilação, o módulo JPMS `casadocodigo.core` deverá ter o `module-info.class` no diretório raiz de seu JAR.

Com um módulo JPMS, conseguimos definir **encapsulamento no nível de pacotes**, escolhendo quais pacotes são ou não exportados e, em consequência, acessíveis por outros módulos.

A JDK modularizada

Um dos grandes avanços do JPMS, disponível a partir da JDK 9, foi a modularização da própria plataforma Java. O JPMS é resultado do projeto *Jigsaw*, criado em 2009, no início do desenvolvimento da JDK 7. Antes do Java 9, todo o código das bibliotecas padrão da JDK ficava em apenas no módulo de runtime: o `rt.jar` .

O estudo inicial do projeto *Jigsaw* agrupou o código já existente da JDK em diferentes módulos. Por exemplo, foi identificado um módulo *base*, que conteria pacotes fundamentais como o `java.lang` e `java.io` ; um módulo Desktop, com as bibliotecas Swing, AWT; além de módulos para APIs como Java Logging, JMX, JNDI.

A análise das dependências entre os módulos identificados na JDK 7 levou à descoberta de ciclos como: o módulo base depende de Logging que depende de JMX que depende de JNDI que depende de Desktop que, por sua vez, depende do módulo base.

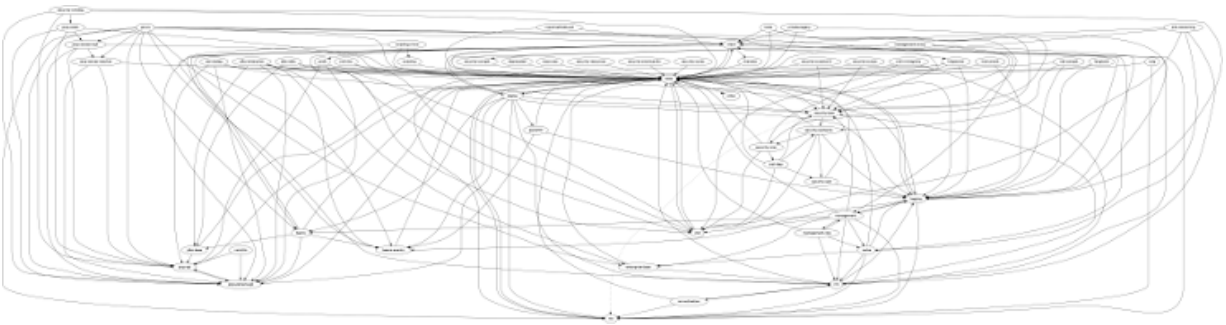

Figura 12.1: Emaranhado de dependências na JDK 7

O código da JDK 8 foi reorganizado para que não houvessem ciclos e dependências indevidas, mesmo que ainda sem um sistema de módulos propriamente dito. Os pacotes que pertenceriam ao módulo `base` não teriam mais dependências a nenhum outro módulo.

Na JDK 9, foram definidos módulos JPMS para cada parte do código da JDK:

- `java.base` , contendo código de pacotes como `java.lang` , `java.math` , `java.text` , `java.io` , `java.net` e `java.nio` ;
- `java.logging` , com a Java Logging API;
- `java.management` , com a Java Managing Extensions (JMX) API;
- `java.naming` , com a Java Naming and Directory Interface (JNDI) API;
- `java.desktop` , contendo código de bibliotecas como Swing, AWT, 2D.

As dependências entre os módulos JPMS da JDK foram organizadas de maneira bem cuidadosa.

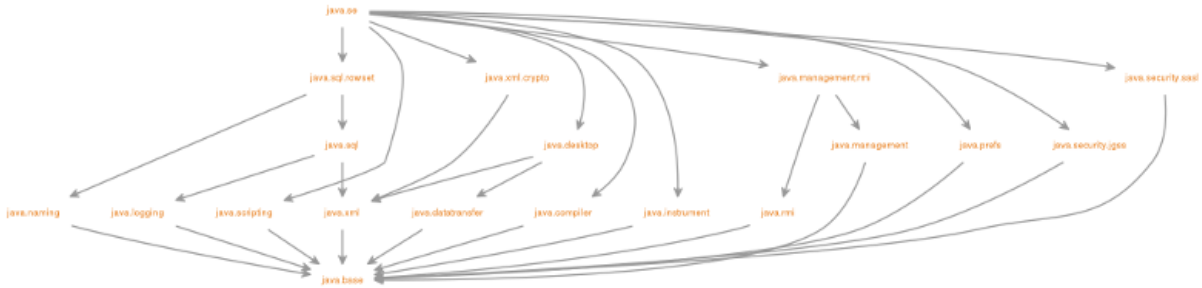

Figura 12.2: Dependências organizadas na JDK 9

Antes da JDK 9, toda aplicação teria disponível todos os pacotes de todas as bibliotecas do Java. Não era possível depender de menos que a totalidade da JDK. A partir da JDK 9, aplicações modularizadas com JPMS podem escolher, no arquivo `module-info.java` , de quais módulos da JDK dependerão. O módulo `java.base` sempre é declarado como dependência, implicitamente.

Usando a JDK modularizada com o Classpath

O Classpath é apenas uma lista de classes. O conceito de módulos JPMS que encapsulam pacotes e dependem de outros módulos não existe para o Classpath. Por outro lado, a partir da versão 9, a JDK é definida em termos de módulos JPMS.

Será que, ao atualizar para a JDK 9, somos obrigados a definir o arquivo `module-info.java` para cada JAR, incluindo o de bibliotecas? A migração de *todos* os módulos iria requerer um trabalho imenso! Impossível de ser feita na prática! Como conciliar o Classpath legado com a JDK modularizada?

Para permitir uma migração mais suave, foi criado o conceito do **unnamed module** (o módulo sem nome), que disponibiliza para o Classpath módulos JPMS. Não bastaria que ficasse disponível apenas o módulo `java.base`, já que aplicações podem depender de código de outros módulos, como o `java.logging` ou o `java.desktop`.

Por isso, o *unnamed module* depende automaticamente do módulo `java.se`, que agrega boa parte dos módulos que estariam disponíveis em versões anteriores, não modularizadas, do Java. São parte do módulo `java.se` os seguintes módulos:

- `java.base`
- `java.compiler`
- `java.datatransfer`
- `java.desktop`
- `java.instrument`
- `java.logging`
- `java.management`
- `java.management.rmi`
- `java.naming`
- `java.prefs`
- `java.rmi`
- `java.scripting`
- `java.security.jgss`
- `java.security.sasl`
- `java.sql`
- `java.sql.rowset`
- `java.xml`
- `java.xml.crypto`

Alguns módulos não são parte do módulo agregador `java.se`. Entre eles, módulos com detalhes específicos da JDK e módulos referentes ao JavaFX.

Para saber mais: módulos Maven vs módulos JPMS

Qual a relação entre módulos Maven e módulos JPMS?

Sander Mak e Paul Bakker afirmam, no livro *Java 9 Modularity* (MAK; BAKKER, 2017), que o Maven não vai muito além de cuidar da configuração da compilação usando o Modulepath e uma mistura de módulos explícitos e automáticos, ambos conceitos que veremos adiante. Mak e Bakker revelam que existem 3 nomes em jogo:

- o nome do módulo JPMS definido no `module-info.java`, usado para ser referenciado por outros módulos no `requires`;
- o nome do projeto Maven definido no `pom.xml`, composto por `groupId`, `artifactId` e `version` e usado para declarar dependências em outros projetos Maven;
- o nome do JAR gerado pelo Maven, que é o artefato entregável final.

Em um projeto Maven que usa o JPMS, são necessários 2 passos: declarar o nome do Maven no `pom.xml` e adicionar um `requires` ao nome do JPMS no `module-info.java`.

No livro *The Java Module System* (PARLOG, 2018), Nicolai Parlog diferencia uma ferramenta de build como o Maven de um sistema de módulos como o JPMS. O Maven tem influência apenas durante a compilação enquanto o JPMS também influencia no runtime de uma aplicação. Para Parlog, o Maven trata de declarar dependências, além de baixá-las, compilar código, rodar testes, gerar artefatos e distribui-los. O JPMS também é usado para declarar dependências, mas também para desacoplar partes de uma aplicação com os plugins da Service Loader API, reforçar o encapsulamento em runtime e, em um uso mais avançado, criar runtimes enxutos por meio das imagens geradas pela ferramenta `jlink`. A intersecção

entre o Maven e o JPMS, portanto, é na declaração de dependências.

12.3 Módulos JPMS no Cotuba

É interessante usarmos o JPMS para termos encapsulamento no nível de pacotes, escolhendo quais podem ou não ser acessados por outros pacotes. Para isso, devemos criar arquivos `module-info.java` em cada um dos módulos, definindo:

- um nome para o módulo;
- quais pacotes serão exportados;
- quais outros módulos são dependências.

Um detalhe importante é que `cotuba-core`, por exemplo, **não** é um nome de módulo JPMS válido, já que hífens não são permitidos, apenas caracteres alfanuméricos e pontos. Por isso, devemos usar um nome como `cotuba.core`.

Vamos iniciar nosso uso de JPMS no módulo Maven `cotuba-core`, que deve exportar para outros módulos:

- o pacote `cotuba.application`, que contém a classe `Cotuba` e as interfaces `ParametrosCotuba` e `RepositorioDeMDs`.
- o pacote `cotuba.domain`, que contém as classes de domínio `Ebook` e `Capitulo` e a enum `FormatoEbook`.
- o pacote `cotuba.plugin`, que contém as SPIs `AoRenderizarHTML`, `AoFinalizarGeracao` e `GeradorEbook`.

O pacote `cotuba.md` não deve ser acessível por outros módulos JPMS. Vamos usar `cotuba.core` como nome do módulo JPMS. Deve ser criado um arquivo `module-info.java` no diretório `src/main/java`, com o seguinte conteúdo:

```
// cotuba/cotuba-core/src/main/java/module-info.java

module cotuba.core {
  exports cotuba.application;
```

```
  exports cotuba.domain;
  exports cotuba.plugin;
}
```

> As IDEs dão suporte à criação desses arquivos. No IntelliJ, no menu *Code > Generate module-info Descriptors*. Já no Eclipse, no menu *Configure > Create module-info.java*.

Esse descritor de módulo definido no arquivo `module-info.java` será compilado em um arquivo `module-info.class` e incluído no JAR do `cotuba.core`.

Para o módulo Maven `cotuba-cli`, vamos definir o módulo JPMS `cotuba.cli`, cujo arquivo `module-info.java` deve ser criado com as seguintes configurações:

```
// cotuba/cotuba-cli/src/main/java/module-info.java

module cotuba.cli {
  requires cotuba.core;
}
```

O `requires` no módulo JPMS `cotuba.cli` indica que há uma dependência em relação ao módulo `cotuba.core`. Não há nenhum pacote exportado para ser utilizado por outros módulos. Não há necessidade de exportar nenhum pacote, já que nenhum outro módulo usará o código de `cotuba.cli`. Apenas o usuário!

Depois de compilarmos o módulo JPMS `cotuba.cli`, serão apresentados erros de compilação para os trechos de código em que usamos dependências indesejadas do próprio Cotuba como `CapituloBuilder` e `RenderizadorMDParaHTML`, além das bibliotecas usadas como `Parser` e `HtmlRenderer`:

```
// cotuba-cli
// cotuba.cli.Main

import cotuba.domain.builder.CapituloBuilder; // error: package
```

```
'cotuba.domain.builder' is declared in module 'cotuba.core', which does
not export it to module 'cotuba.cli'
import cotuba.md.RenderizadorMDParaHTML; // error: package 'cotuba.md' is
declared in module 'cotuba.core', which does not export it to module
'cotuba.cli'

import org.commonmark.parser.Parser; // error: package
'org.commonmark.parser' is declared in module 'org.commonmark', but module
'cotuba.cli' does not read it
import org.commonmark.renderer.html.HtmlRenderer; // error: package
'org.commonmark.renderer.html' is declared in module 'org.commonmark', but
module 'cotuba.cli' does not read it

public class Main {

  public static void main(String[] args) {

    // error: cannot find symbol
    CapituloBuilder builder = new CapituloBuilder();
    System.out.println(builder);

    // error: cannot find symbol
    RenderizadorMDParaHTML md = new RenderizadorMDParaHTML();
    System.out.println(md);

    // error: cannot find symbol
    Parser parser = new Parser.Builder().build();
    System.out.println(parser);

    // error: cannot find symbol
    HtmlRenderer htmlRenderer =
      new HtmlRenderer.Builder().build();
    System.out.println(htmlRenderer);

    // código omitido...

  }

}
```

Pronto! Por meio do JPMS, detalhes internos do módulo JPMS `cotuba.core` e dependências transitivas das bibliotecas utilizadas estão escondidos. O **encapsulamento foi fortalecido** no nível de pacotes!

Podemos remover, da classe `Main` do módulo `cotuba.cli`, os usos indevidos de `CapituloBuilder`, `RenderizadorMDParaHTML`, `Parser` e `HtmlRenderer`:

```
// cotuba-cli
// cotuba.cli.Main

// código omitido...

import cotuba.domain.builder.CapituloBuilder;
import cotuba.md.RenderizadorMDParaHTML;

import org.commonmark.parser.Parser;
import org.commonmark.renderer.html.HtmlRenderer;

public class Main {

  public static void main(String[] args) {

    CapituloBuilder builder = new CapituloBuilder();
    System.out.println(builder);

    RenderizadorMDParaHTML md = new RenderizadorMDParaHTML();
    System.out.println(md);

    Parser parser = new Parser.Builder().build();
    System.out.println(parser);

    HtmlRenderer htmlRenderer = new HtmlRenderer.Builder().build();
    System.out.println(htmlRenderer);

    // código omitido...

  }
```

```
}
```

Em seguida, vamos criar o arquivo `module-info` no módulo Maven `cotuba-pdf`, definindo seu nome como `cotuba.pdf`. Deve ser exportado o pacote `cotuba.pdf`, que contém a classe `GeradorPDF`. Também deve ser declarada a dependência ao módulo JPMS `cotuba.core`:

```
// cotuba/cotuba-pdf/src/main/java/module-info.java

module cotuba.pdf {
  exports cotuba.pdf;

  requires cotuba.core;
}
```

Também precisamos criar o `module-info.java` para o módulo Maven `cotuba-epub`, usando o nome `cotuba.epub`. Deve ser declarada a dependência ao módulo JPMS `cotuba.core` e exportado o pacote `cotuba.epub`:

```
// cotuba/cotuba-epub/src/main/java/module-info.java

module cotuba.epub {
  exports cotuba.epub;

  requires cotuba.core;
}
```

Finalmente, vamos transformar o módulo Maven `cotuba-html` em um módulo JPMS chamado `cotuba.html`, criando um `module-info.java` que declara a dependência ao módulo JPMS `cotuba.core` e exporta o pacote `cotuba.html`:

```
// cotuba/cotuba-html/src/main/java/module-info.java

module cotuba.html {
  exports cotuba.html;
```

```
  requires cotuba.core;
}
```

Mas nem tudo está bem! Depois de configurar o JPMS nos módulos, começaram a ocorrer alguns erros de compilação:

- em `cotuba.core`, no uso de classes da biblioteca CommonMark como `Parser`, `Heading` e `HtmlRenderer` em `RenderizadorMDParaHTML`;
- em `cotuba.cli`, no uso de classes da biblioteca Apache Commons CLI como `CommandLine`, `Options` e `HelpFormatter` em `LeitorOpcoesCLI`;
- em `cotuba.pdf`, no uso das classes da biblioteca iText pdfHTML, como `PdfWriter` e `PdfDocument`, em `GeradorPDF`;
- em `cotuba.epub`, no uso das classes da biblioteca Epublib, como `Book` e `EpubWriter`, em `GeradorEPUB`.

Como será que o JPMS funciona ao usarmos bibliotecas de terceiros?

12.4 Módulos automáticos do JPMS

A maioria das bibliotecas, nas versões que utilizamos, não define um arquivo `module-info.class` em seu JARs, ou seja, não são modularizadas. É o caso das bibliotecas CommonMark e Apache Commons CLI, por exemplo.

Mas um módulo JPMS precisa declarar em seu `module-info.java` que depende dos módulos dessas bibliotecas. O que colocar no `requires` ?

Para permitir que JARs modularizados dependam de JARs não modularizados, suavizando o esforço de migração, foi criado o conceito dos **Automatic Modules** (módulos automáticos) no JPMS. Um *Automatic Module*:

- tem seu nome derivado do JAR, trocando hifens por pontos e removendo a versão;
- tem todos os seus pacotes exportados;
- tem todos os outros módulos declarados como dependências.

Uma biblioteca não modularizada pode configurar o nome de seu *Automatic Module* definindo a propriedade `Automatic-Module-Name` no arquivo `META-INF/MANIFEST.MF` . O JAR da biblioteca CommonMark, por exemplo, define um `Automatic-Module-Name` com o valor `org.commonmark` .

Podemos usar o comando `jar` para descobrir o nome do *Automatic Module* criado a partir de um JAR. Basta executar o comando a seguir:

```
jar --file commons-cli-1.4.jar --describe-module
```

O JAR do Apache Commons CLI pode ser encontrado no repositório local do Maven, o subdiretório `.m2/repository` do *home* do seu usuário, no subdiretório `commons-cli/commons-cli/1.4` .

Como resposta do comando anterior, obteríamos algo como:

```
No module descriptor found. Derived automatic module.

commons.cli@1.4 automatic
requires java.base mandated
contains org.apache.commons.cli
```

Repare, na primeira linha da saída anterior, que temos `commons.cli@1.4 automatic`. A versão é removida do nome gerado para o módulo JPMS automático e, portanto, teríamos o nome `commons.cli` para a biblioteca Apache Commons CLI.

Para as bibliotecas usadas por nossa solução como um todo, teríamos os seguintes *Automatic Modules*:

- `epublib.core` para o `epublib-core-3.1.jar`
- `html2pdf` para o `html2pdf-2.0.0.jar`, parte do iText pdfHTML
- `io` para o `io-7.1.0.jar`, parte do iText pdfHTML
- `kernel` para o `kernel-7.1.0.jar`, parte do iText pdfHTML
- `layout` para o `layout-7.1.0.jar`, parte do iText pdfHTML
- `jsoup` para o `jsoup-1.11.2.jar`, usado nos plugins de tema da Paradizo e de estatísticas da Cognitio
- `commons.cli` para o `commons-cli-1.4.jar`, conforme vimos no comando anterior
- `org.commonmark` para o `commonmark-0.11.0.jar`, definido pela propriedade `Automatic-Module-Name`

Módulos JPMS automáticos no Cotuba

Vamos definir o módulo JPMS automático da biblioteca CommonMark como dependência do módulo JPMS `cotuba.core`:

```
// cotuba/cotuba-core/src/main/java/module-info.java

module cotuba.core {
  exports cotuba.application;
  exports cotuba.plugin;
  exports cotuba.domain;
```

```
  requires org.commonmark; // inserido
}
```

Com o código anterior, a classe `RenderizadorMDParaHTML` do módulo JPMS `cotuba.core` deve voltar a ser compilada com sucesso, já que passamos a indicar, pelo nome JPMS automático, que a biblioteca CommonMark é utilizada.

Devemos fazer algo semelhante para o módulo JPMS `cotuba.cli`:

```
// cotuba/cotuba-cli/src/main/java/module-info.java

module cotuba.cli {
  requires cotuba.core;

  requires commons.cli; // inserido
}
```

A classe `LeitorOpcoesCLI` do módulo JPMS `cotuba.cli` deve ser compilada com sucesso novamente. Em seguida, devemos fazer algo semelhante para o módulo JPMS `cotuba.pdf`:

```
// cotuba/cotuba-pdf/src/main/java/module-info.java

module cotuba.pdf {
  exports cotuba.pdf;
  requires cotuba.core;

  requires html2pdf; // inserido
  requires io; // inserido
  requires kernel; // inserido
  requires layout; // inserido
}
```

Depois disso, a compilação de `GeradorPDF` deve voltar a ser bem-sucedida! Não podemos nos esquecer de declarar as dependências a bibliotecas no módulo JPMS `cotuba.epub`:

```
// cotuba/cotuba-epub/src/main/java/module-info.java
```

```
module cotuba.epub {
  exports cotuba.epub;
  requires cotuba.core;

  requires epublib.core; // inserido
}
```

O módulo `cotuba.html` não usa nenhuma biblioteca de terceiros, apenas código incluído no módulo `java.base`.

Pronto! O código deve compilar com sucesso. Porém, ao tentarmos executar a classe `Main` de `cotuba.cli` obteremos a seguinte exceção em runtime:

```
Exception in thread "main" java.util.ServiceConfigurationError:
  cotuba.plugin.AoRenderizarHTML:
    module cotuba.core does not declare `uses`
...
at java.base/java.util.ServiceLoader
    .load(ServiceLoader.java:1691)
at cotuba.core/cotuba.plugin.AoRenderizarHTML
    .renderizou(AoRenderizarHTML.java:11)
at cotuba.core/cotuba.md.RenderizadorMDParaHTML
    .renderizaParaHTML(RenderizadorMDParaHTML.java:63)
...
```

Quando usamos a SPIs e Service Providers com JPMS, precisamos fazer alguns ajustes!

12.5 Service Loader API com JPMS

A exceção `java.util.ServiceConfigurationError` ocorre porque um módulo JPMS precisa declarar explicitamente as interfaces que são *Service Provider Interface* (SPI), cujas implementações poderão ser carregadas com um *Service Loader*.

Repare na mensagem de erro:

```
module cotuba.core does not declare `uses`
```

Devemos declarar o uso de uma SPI no `module-info.java`, por meio da palavra-chave `uses`. O módulo JPMS `cotuba.core` deve declarar o uso das SPIs `AoRenderizarHTML`, `AoFinalizarGeracao` e `GeradorEbook` da seguinte maneira:

```
// cotuba/cotuba-core/src/main/java/module-info.java

module cotuba.core {

  // código omitido...

  uses cotuba.plugin.AoRenderizarHTML; // inserido
  uses cotuba.plugin.AoFinalizarGeracao; // inserido
  uses cotuba.plugin.GeradorEbook; // inserido

}
```

Já um *Service Provider*, que implementa uma SPI, não precisa mais do arquivo com o nome da SPI no `META-INF/services`. Porém, pode fazer sentido se o módulo for usado com o Classpath. Basta usarmos, no `module-info.java` as palavras-chave `provides` com o nome da SPI e `with` com o nome da implementação.

Para o módulo JPMS `cotuba.pdf`:

```
// cotuba/cotuba-pdf/src/main/java/module-info.java

import cotuba.pdf.GeradorPDF; // inserido

module cotuba.pdf {

  // código omitido...

  provides cotuba.plugin.GeradorEbook with GeradorPDF; // inserido

}
```

Para o módulo JPMS `cotuba.epub`:

```
// cotuba/cotuba-epub/src/main/java/module-info.java

import cotuba.epub.GeradorEPUB; // inserido

module cotuba.epub {

  // código omitido...

  provides cotuba.plugin.GeradorEbook with GeradorEPUB; // inserido

}
```

Para o módulo JPMS `cotuba.html`:

```
// cotuba/cotuba-html/src/main/java/module-info.java

import cotuba.html.GeradorHTML; // inserido

module cotuba.html {

  // código omitido...

  provides cotuba.plugin.GeradorEbook with GeradorHTML; // inserido

}
```

Mantenha o arquivo `META-INF/services/cotuba.plugin.GeradorEbook` nos módulos JPMS geradores de PDFs, EPUBs e HTMLs. Esse arquivo ainda será usado quando usarmos o módulo `cotuba-pdf` no Classpath.

Ao executarmos novamente a classe `Main` de `cotuba.cli`, os ebooks devem ser gerados com sucesso!

12.6 O Modulepath

Ao definirmos o arquivo `module-info.java` nos módulos do Cotuba, o encapsulamento é reforçado apenas em tempo de compilação. No runtime, ao usarmos a configuração `-cp` do comando `java`, estamos configurando onde o Classpath deve buscar arquivos `.class`.

Ao executarmos um JAR modularizado com JPMS usando o Classpath, as vantagens de encapsulamento em nível de pacotes não são reforçadas.

Podemos testar isso executando o código que vimos anteriormente que, no módulo JPMS `cotuba.pdf`, usava a Reflection API e a falta de encapsulamento do Classpath para instanciar classes que não fazem partes de suas dependências, como `Main` do módulo JPMS `cotuba.cli` e `Options` da biblioteca Apache Commons CLI:

```
// cotuba-pdf
// cotuba.pdf.GeradorPDF

// código omitido...

try {

  Class mainClass = Class.forName("cotuba.cli.Main");
  Object main = mainClass.getDeclaredConstructor().newInstance();
  System.out.println(main);

  Class optionsClass =
      Class.forName("org.apache.commons.cli.Options");
  Object options =
      optionsClass.getDeclaredConstructor().newInstance();
  System.out.println(options);

} catch (ClassNotFoundException | InstantiationException|
```

```
        IllegalAccessException | NoSuchMethodException |
        InvocationTargetException ex) {
  throw new IllegalStateException(ex);
}

// código omitido...
```

Ao realizarmos o build pelo Maven e executarmos o `cotuba.sh`, tudo funcionará normalmente. Ao gerarmos um PDF, teremos na saída algo como:

```
cotuba.cli.Main@20398b7c
[ Options: [ short {} ] [ long {} ]
Arquivo gerado com sucesso: book.pdf
```

> Provavelmente o código apresentará um erro de runtime ao ser executado a partir de uma IDE, que já deve realizar as configurações necessárias.

As restrições do JPMS só são reforçadas se usarmos um **Modulepath**. Para definir o Modulepath com diretórios que contém JARs modularizados, devemos usar a opção `--module-path` nos comandos de compilação (`javac`) e execução (`java`). A compilação feita pelo `maven-compiler-plugin` já cuida desses detalhes para o comando `javac`.

Porém, precisamos modificar nosso script `cotuba.sh` para utilizar a opção `--module-path`, passando o diretório `libs` como fonte dos JARs. Além disso, precisamos usar a opção `--module` para dizer qual é o módulo JPMS e a classe que contém o método `main` que queremos executar:

```
# cotuba-cli/src/scripts/cotuba.sh

#!/bin/bash
java -cp "libs/*" cotuba.cli.Main "$@"
java --module-path libs --module cotuba.cli/cotuba.cli.Main "$@"
```

Depois de um novo build com `mvn install` e de descompactar o ZIP gerado, ao executarmos a geração de PDF novamente com o `cotuba.sh`, teremos uma exceção em runtime parecida com a seguinte:

```
java.lang.IllegalAccessException: class cotuba.pdf.GeradorPDF (in module
cotuba.pdf)
  cannot access class cotuba.cli.Main (in module cotuba.cli)
  because module cotuba.cli does not export cotuba.cli to module
cotuba.pdf
```

Note que, com o Modulepath, o encapsulamento foi reforçado e foi detectado que:

- a classe `cotuba.cli.Main`, que queremos instanciar pela Reflection API, tem como origem o módulo JPMS `cotuba.cli`;
- a classe que está sendo executada, a `cotuba.pdf.GeradorPDF`, tem o módulo `cotuba.pdf` como origem;
- o módulo JPMS `cotuba.cli` não exporta o pacote `cotuba.cli` para o módulo JPMS `cotuba.pdf`.

Portanto, ocorre uma `IllegalAccessException`, indicando que não é possível fazer o uso da Reflection API para entre essas classes. Vamos remover o uso da classe `Main` de `GeradorPDF`:

```
// cotuba-pdf
// cotuba.pdf.GeradorPDF

// código omitido...

try {

  Class mainClass = Class.forName("cotuba.cli.Main");
  Object main = mainClass.getDeclaredConstructor().newInstance();
  System.out.println(main);

  Class optionsClass =
      Class.forName("org.apache.commons.cli.Options");
  Object options =
      optionsClass.getDeclaredConstructor().newInstance();
```

```
  System.out.println(options);

} catch (ClassNotFoundException | InstantiationException|
        IllegalAccessException | NoSuchMethodException |
        InvocationTargetException ex) {
  throw new IllegalStateException(ex);
}

// código omitido...
```

Ao executarmos o `cotuba.sh`, mesmo com o `Modulepath` configurado, não teríamos nenhuma exceção:

```
[ Options: [ short {} ] [ long {} ]
Arquivo gerado com sucesso: book.pdf
```

O resultado do `toString` da classe `Options` foi exibido e o ebook foi gerado com sucesso no formato PDF. Estranho, não? Será que isso indica que o encapsulamento não é tão forte assim?

Na verdade, um módulo JPMS automático, criado para bibliotecas que não declaram um `module-info.java`, é **aberto** para *reflection*, permitindo o uso da Reflection API em qualquer tipo público.

Porém, se tentarmos executar a geração de um PDF a partir da UI Web do Cotuba, teremos uma exceção de `ClassNotFoundException`. Isso acontece porque a biblioteca Apache Commons CLI não faz parte das dependências do módulo Maven `cotuba-web`. Por isso, é importante removermos, da classe `GeradorPDF`, a instanciação de `Options`:

```
// cotuba-pdf
// cotuba.pdf.GeradorPDF

// código omitido...

try {

  Class optionsClass = Class.forName("org.apache.commons.cli.Options");
  Object options = optionsClass.getDeclaredConstructor().newInstance();
```

```
  System.out.println(options);

}catch(ClassNotFoundException|InstantiationException|
        IllegalAccessException|NoSuchMethodException|
        InvocationTargetException ex){
  throw new IllegalStateException(ex);
}

// código omitido...
```

Executando novamente a geração de PDFs pela UI Web, tudo deve funcionar. O mesmo deve acontecer para EPUBs!

Ainda não transformamos o `cotuba-web` em um módulo JPMS. Vamos lá?

12.7 JPMS e o Spring Boot

Por enquanto, o módulo Maven `cotuba-web` não foi definido como um módulo JPMS. E tudo funciona sem problemas! O mesmo deve acontecer para os plugins `tema-paradizo` e `estatisticas-ebook` que, se não forem definidos como módulos JPMS, não precisam sofrer alterações!

É interessante definirmos um `module-info.java` para a UI Web do Cotuba, configurando seu nome como `cotuba.web` e declarando o módulo JPMS `cotuba.core` como dependência:

```
// cotuba/cotuba-web/src/main/java/module-info.java

module cotuba.web {

  requires cotuba.core;

}
```

Ao efetuarmos o build, teremos uma série de erros de compilação em diferentes classes. Precisamos declarar como dependências:

- `java.persistence` para que anotações do JPA como `@Entity` e `@Table`
- `java.validation` para anotações do Bean Validation como `@Min` e `@Size`
- `spring.context` para anotações do Spring como `@Service` e `@Controller`
- `spring.web` para anotações do Spring MVC como `@GetMapping` e `@PathVariable`
- `spring.data.jpa` para a interface `JpaRepository`
- `spring.boot` para a classe `SpringApplication`
- `spring.boot.autoconfigure` para a anotação `@SpringBootApplication`
- `org.apache.tomcat.embed.core` para a interface `HttpServletResponse`

Depois de todas essas declarações, o arquivo `module-info.java` do módulo JPMS `cotuba.web` ficaria da seguinte forma:

```
// cotuba/cotuba-web/src/main/java/module-info.java

module cotuba.web {
  requires cotuba.core;

  requires java.persistence; // inserido
  requires java.validation; // inserido

  requires spring.context; // inserido
  requires spring.web; // inserido

  requires spring.data.jpa; // inserido

  requires spring.boot; // inserido
  requires spring.boot.autoconfigure; // inserido

  requires org.apache.tomcat.embed.core; // inserido

}
```

> É interessante notar que todas as dependências declaradas anteriormente são módulos JPMS automáticos, que declaram a propriedade `Automatic-Module-Name` no arquivo `META-INF/MANIFEST.MF` de seus JARs.

Podemos fazer o build, a partir do supermódulo Maven `cotuba`:

```
# cotuba

mvn clean install
```

O build será bem-sucedido:

```
[INFO] Reactor Summary for cotuba 0.0.1-SNAPSHOT:
[INFO]
[INFO] cotuba ............................. SUCCESS [  0.060 s]
```

```
[INFO] cotuba-core ........................ SUCCESS [  1.030 s]
[INFO] cotuba-pdf ......................... SUCCESS [  0.150 s]
[INFO] cotuba-epub ........................ SUCCESS [  0.090 s]
[INFO] cotuba-cli ......................... SUCCESS [  0.904 s]
[INFO] cotuba-web ......................... SUCCESS [  0.787 s]
[INFO] ------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------
[INFO] Total time:  3.161 s
```

Podemos tentar executar a UI Web do Cotuba com o comando:

```
# cotuba

java -jar cotuba-web/target/cotuba-web-0.0.1-SNAPSHOT.jar
```

Pronto! Tudo deve funcionar! Podemos editar o conteúdo dos capítulos e gerar PDFs e EPUBs!

Abrindo módulos JPMS para reflexão

Executamos a UI Web do Cotuba sem problemas! O Spring Boot gera um *fat JAR* que contém todas as classes de todos os JARs utilizados pela aplicação. No nosso caso, o `cotuba-web-0.0.1-SNAPSHOT.jar` tem 46 MB! Não precisamos usar nem o Classpath nem o Modulepath, já que tudo está incluído dentro do mesmo JAR.

Entretanto, ao executarmos a classe `CotubaWebApplication` pela IDE, pode ser que tenhamos problemas. Por exemplo, no IntelliJ, ao tentarmos subir a aplicação, veremos uma exceção parecida com a seguinte:

```
org.springframework.cglib.core.CodeGenerationException:
  java.lang.IllegalAccessException-->module cotuba.web does not open
    cotuba.web to unnamed module @2e4b8173
```

O Spring Boot faz um uso massivo da Reflection API mas os módulos JPMS proíbem esse tipo de acesso por padrão. Perceba que se trata de um erro relacionado à geração de código pela *cglib* (Code Generation Library), uma biblioteca que manipula e cria

classes em runtime e é umas das responsáveis pelas magias de ferramentas como o Spring.

Para que o Spring Context, por exemplo, acesse via *reflection* as classes do pacote `cotuba.web` do módulo JPMS de mesmo nome, poderíamos usar um `opens` da seguinte maneira:

```
// cotuba/cotuba-web/src/main/java/module-info.java

module cotuba.web {

  // código omitido...

  opens cotuba.web to spring.context;

}
```

Mas não teríamos sucesso. Continuaria acontecendo a mesma exceção...

Repare, na mensagem de erro anterior, que indica que precisaríamos abrir o pacote `cotuba.web` para o *unnamed module*, o módulo sem nome. Como a própria JDK é modularizada, foi necessário criar um módulo JPMS de compatibilidade, que inclui todo o conteúdo do Classpath. Esse módulo não possui um descritor nem um nome e, por isso, é chamado de *unnamed module*. E nossas IDEs, por padrão, usam o Classpath para executar as aplicações.

Poderíamos, então, abrir o pacote `cotuba.web` para o qualquer módulo, incluindo o *unnamed module*, da seguinte maneira:

```
// cotuba/cotuba-web/src/main/java/module-info.java

module cotuba.web {

  // código omitido...

  opens cotuba.web to spring.context;
  opens cotuba.web;
```

```
}
```

Pronto! Ou quase... Teríamos um erro ligeiramente diferente:

```
java.lang.IllegalStateException: Unable to load cache item
  at org.springframework.cglib.core.internal.LoadingCache
    .createEntry(LoadingCache.java:79)
  ~[spring-core-5.3.15.jar:5.3.15]
  ...
  at cotuba.web/cotuba.web.CotubaWebApplication
    .main(CotubaWebApplication.java:10) ~[classes/:na]

Caused by: java.lang.IllegalAccessError:
  class cotuba.web
    .CotubaWebApplication$$EnhancerBySpringCGLIB$$7d8a29c5
        (in module cotuba.web)
  cannot access class org.springframework.cglib.core.ReflectUtils
      (in unnamed module @0x2e4b8173)
  because module cotuba.web does not read
      unnamed module @0x2e4b8173
  at cotuba.web/cotuba.web
      .CotubaWebApplication$$EnhancerBySpringCGLIB$$7d8a29c5
      .CGLIB$STATICHOOK1(<generated>)
  ~[classes/:na]
```

Ainda se trata de um erro relacionado à geração de código pela *cglib*. Pelo erro, é possível notar que se trata de algo relacionado ao `spring-core-5.3.15.jar`. Vamos declarar como dependência o `spring.core` módulo JPMS automático criado para essa biblioteca:

```
// cotuba/cotuba-web/src/main/java/module-info.java

module cotuba.web {

  // código omitido...

  opens cotuba.web;

  requires spring.core; // inserido
```

```
}
```

Ao executarmos novamente a aplicação, teríamos outro erro bem parecido:

```
Caused by: java.lang.IllegalAccessError:
  class cotuba.web.CotubaWebApplication$$EnhancerBySpringCGLIB$$34e91233
(in module cotuba.web)
  cannot access class org.springframework.beans.BeansException (in unnamed
module @0x2ca923bb)
  because module cotuba.web does not read unnamed module @0x2ca923bb
```

Trata-se de algo relacionado ao módulo JPMS automático `spring.beans`:

```
// cotuba/cotuba-web/src/main/java/module-info.java

module cotuba.web {

  // código omitido...

  opens cotuba.web;

  requires spring.core;
  requires spring.beans; // inserido

}
```

Ainda não será dessa vez... O erro muda:

```
org.springframework.beans.factory.BeanCreationException:
  Error creating bean with name 'flywayInitializer' defined in
      class path resource
  [org/springframework/boot/autoconfigure/flyway/
      FlywayAutoConfiguration$FlywayConfiguration.class]:
  Invocation of init method failed; nested exception is
      org.flywaydb.core.api.FlywayException:
  Unable to obtain inputstream for resource:
      db/migration/V001__cria_tabela_livros.sql
```

A biblioteca Flyway, responsável por executar scripts SQL de migração de dados como o `V001__cria_tabela_livros.sql` referenciado no erro, precisa que recursos sejam abertos. Mas não podemos fazer isso colocando um `opens` em algum dos pacotes.

Para permitir acesso a recursos, além de classes, precisamos declarar o módulo JPMS todo como aberto para reflexão adicionado `open` antes da declaração:

```
// cotuba/cotuba-web/src/main/java/module-info.java

open module cotuba.web { // modificado

  // código omitido...

  opens cotuba.web;

  requires spring.core;
  requires spring.beans;

}
```

Estamos quase, mas não foi dessa vez. Teríamos o seguinte erro:

```
Caused by: java.lang.IllegalAccessError: superinterface check
    failed:
  class cotuba.web.domain.Capitulo$HibernateProxy$IvqEP1Bi
      (in module cotuba.web)
  cannot access class org.hibernate.proxy.ProxyConfiguration
      (in unnamed module @0x72d6b3ba)
  because module cotuba.web does not read
      unnamed module @0x72d6b3ba
  ...
  at net.bytebuddy.utility.dispatcher
      .JavaDispatcher$ProxiedInvocationHandler
      .invoke(JavaDispatcher.java:1142)
  ~[byte-buddy-1.11.22.jar:na]
```

Trata-se de algo relacionado à geração de código em runtime pela biblioteca Byte Buddy, umas das responsáveis pelas magias do

Hibernate. Podemos corrigir isso colocando, como dependência de `cotuba.web`, o módulo JPMS automático `org.hibernate.orm.core`:

```
// cotuba/cotuba-web/src/main/java/module-info.java

open module cotuba.web {

  // código omitido...

  requires spring.core;
  requires spring.beans;
  requires org.hibernate.orm.core; // inserido

}
```

Ufa! Agora sim! A classe `CotubaWebApplication` da UI Web do Cotuba pode ser executado com sucesso pela IDE! A edição de capítulos e a geração de PDFs e EPUBs deve funcionar!

12.8 O que aprendemos?

Neste capítulo, foram revelados os problemas do Classpath e como o JPMS pretende resolvê-los. Os diversos módulos Maven do Cotuba foram migrados para módulos JPMS, por meio da definição de arquivos `module-info.java`. Vimos como declarar dependências a JARs não modularizados por meio dos módulos automáticos do JPMS. Também estudamos como integrar a *Service Loader* API com o JPMS e usamos o Modulepath para reforçar o encapsulamento também no runtime. Finalmente, migramos uma aplicação Spring Boot para usar o JPMS e aprendemos como abrir um módulo JPMS para reflexão. Bastante coisa!

No próximo capítulo, aprenderemos um nome para o estilo arquitetural que fomos adotando progressivamente nesse livro!

O código deste capítulo pode ser encontrado em: https://github.com/alexandreaquiles/solid-na-pratica/tree/cap12-jpms

Figura 12.3: Vídeo do capítulo

CAPÍTULO 13

Arquitetura Hexagonal: uma arquitetura centrada no domínio

Uma das arquiteturas mais comuns em aplicações corporativas é a arquitetura em 3 camadas. Nessa arquitetura, a camada de Apresentação depende da camada de Negócio que, por sua vez, depende da camada de Persistência. Sob uma ótica dos princípios SOLID, há uma violação do DIP, já que há uma dependência de uma camada de alto nível (Negócio) em direção a uma camada de baixo nível (Persistência).

Podemos inverter as dependências por meio de abstrações fornecidas pela camada de Negócio e implementadas pela camada de Persistência. Ao invertermos as dependências, Persistência depende do Negócio, e não o contrário. Alto nível não depende mais de baixo nível e, portanto, o DIP é respeitado.

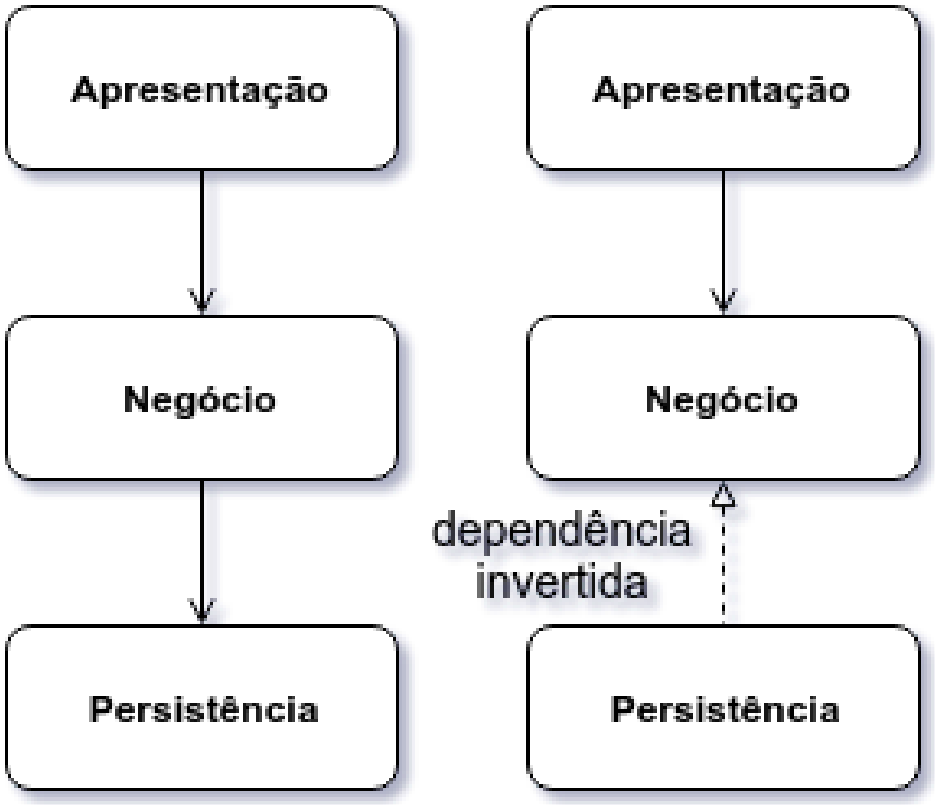

Figura 13.1: 3 Camadas x Dependências Invertidas

Todas as setas do diagrama anterior passam a apontar para o Negócio. Dessa forma, teríamos uma arquitetura centrada na lógica de negócio e o modelo de domínio passaria a ser o núcleo da aplicação. Essa organização centrada no domínio é uma maneira de estruturar o código de uma aplicação que é comumente chamada de *Arquitetura Hexagonal*.

13.1 Hexágonos (ou Conectores & Adaptadores)

Alistair Cockburn, em seu artigo *Hexagonal Architecture* (COCKBURN, 2005), define um estilo arquitetural que separa o código em duas partes: a de dentro e a de fora. A parte interna contém a lógica de Negócio. Já a externa, contém detalhes de implementação como UI e Persistência.

A parte interna, de Negócio, fornece uma API que é usada e/ou implementada pela parte externa. O intuito, no fim das contas, é organizar a aplicação apartando o código relacionado a domínio do código mais técnico, relacionado a mecanismos de entrega. Dessa maneira, é possível guiar o núcleo da aplicação por diferentes clientes: além da UI, as regras de negócio podem ser usadas por scripts, outros programas e até por testes automatizados.

A mesma arquitetura também é chamada por Alistair Cockburn de **Ports & Adapters**, algo como Conectores e Adaptadores. A API fornecida pela aplicação é análoga aos *ports* (ou conectores) de dispositivos eletrônicos, como uma porta USB, que permitem que dispositivos externos sejam plugados. Cada conector é ligado a outros dispositivos por um ou mais adaptadores.

Um conector de UI fornecido pelo núcleo da aplicação pode ser usado por dispositivos como uma UI de linha de comando, uma UI Web, um teste automatizado, um reconhecedor de voz ou um *chat bot*.

Já um conector de Persistência pode ser usado por um Banco de Dados relacional, um Banco de Dados não relacional, um Banco de Dados em memória em testes automatizados ou por um Web Service externo, como o Firebase.

Um adaptador é um trecho de código que implementa um conector, fazendo a ponte com um dispositivo específico. Nos termos de Alistair Cockburn, um adaptador faz a tradução do protocolo do conector para o de um dispositivo.

O hexágono da Arquitetura Hexagonal é uma metáfora visual que traz a ideia de uma parte interna e uma parte externa. Além disso, um hexágono não é unidimensional como um desenho de camadas. Os seis lados do hexágono remetem aos vários conectores de entrada e saída.

Os engenheiros da Netflix Damir Svrtan e Sergii Makagon afirmam no artigo *Ready for changes with Hexagonal Architecture* (SVRTAN; MAKAGON, 2020) afirmam que a ideia da Arquitetura Hexagonal é colocar entradas e saídas nas bordas de um design de código. Para os autores, a lógica de negócios não deve depender da exposição de uma API REST ou GraphQL, nem de onde obtemos dados, seja um Banco de Dados, uma API de um Microservice exposta por gRPC ou REST, ou um simples arquivo CSV. Com essa organização é possível isolar o núcleo da lógica de nossa aplicação de preocupações externas. Dessa maneira, podemos alterar facilmente os detalhes da fonte de dados sem um impacto significativo e sem reescrever grande parte do código. Svrtan e Makagon afirmam: *"Uma das principais vantagens que vimos ao ter uma aplicação com limites claros é a nossa estratégia de teste: a maioria dos nossos testes pode verificar nossa lógica de negócios sem depender de protocolos que podem mudar facilmente."*

13.2 Arquitetura Hexagonal no Cotuba

O Cotuba tem 5 módulos:

- `cotuba-core` , de alto nível, contém a lógica de negócio.
- `cotuba-pdf` , de baixo nível, gera ebooks no formato PDF.
- `cotuba-epub` , de baixo nível, gera ebooks no formato EPUB.
- `cotuba-html` , de baixo nível, gera um site HTML.
- `cotuba-cli` , de baixo nível, define uma UI de linha de comando.
- `cotuba-web` , de baixo nível, define uma UI Web.

O módulo central, o núcleo da aplicação, é o `cotuba-core` . Esse módulo renderiza arquivos `.md` para `.html` , chama o gerador de ebook no formato apropriado e permite ações pós-renderização e pós-geração. Alguns dos conectores (ou *ports*) fornecidos são:

- `ParametrosCotuba` , um conector usado para receber parâmetros como o formato de ebook a ser gerado e o arquivo de saída.
- `RepositorioDeMDs` , um conector usado como fonte dos MDs.
- `GeradorEbook` , um conector usado para a geração de ebooks em diferentes formatos.

O módulo `cotuba-cli` torna o Cotuba acessível via Terminal, fornecendo adaptadores para os conectores `ParametrosCotuba` , que obtém parâmetros das opções de linha de comando, e `RepositorioDeMDs` , que lê os MDs de um diretório.

O módulo `cotuba-web` torna o Cotuba acessível via Navegador. O adaptador de `ParametrosCotuba` desse módulo obtém o formato do ebook a ser gerado da URL e o arquivo de saída de um diretório aleatório. Já o adaptador de `RepositorioDeMDs` lê os MDs do BD.

O módulo `cotuba-pdf` fornece um adaptador para `GeradorEbook` que usa a biblioteca iText pdfHTML para gerar um PDF a partir de

HTMLs.

O módulo `cotuba-epub` fornece um adaptador para `GeradorEbook` que gera um EPUB a partir dos HTMLs por meio da biblioteca Epublib.

O módulo `cotuba-html` fornece um adaptador para `GeradorEbook` que salva os arquivos HTML renderizado em um diretório.

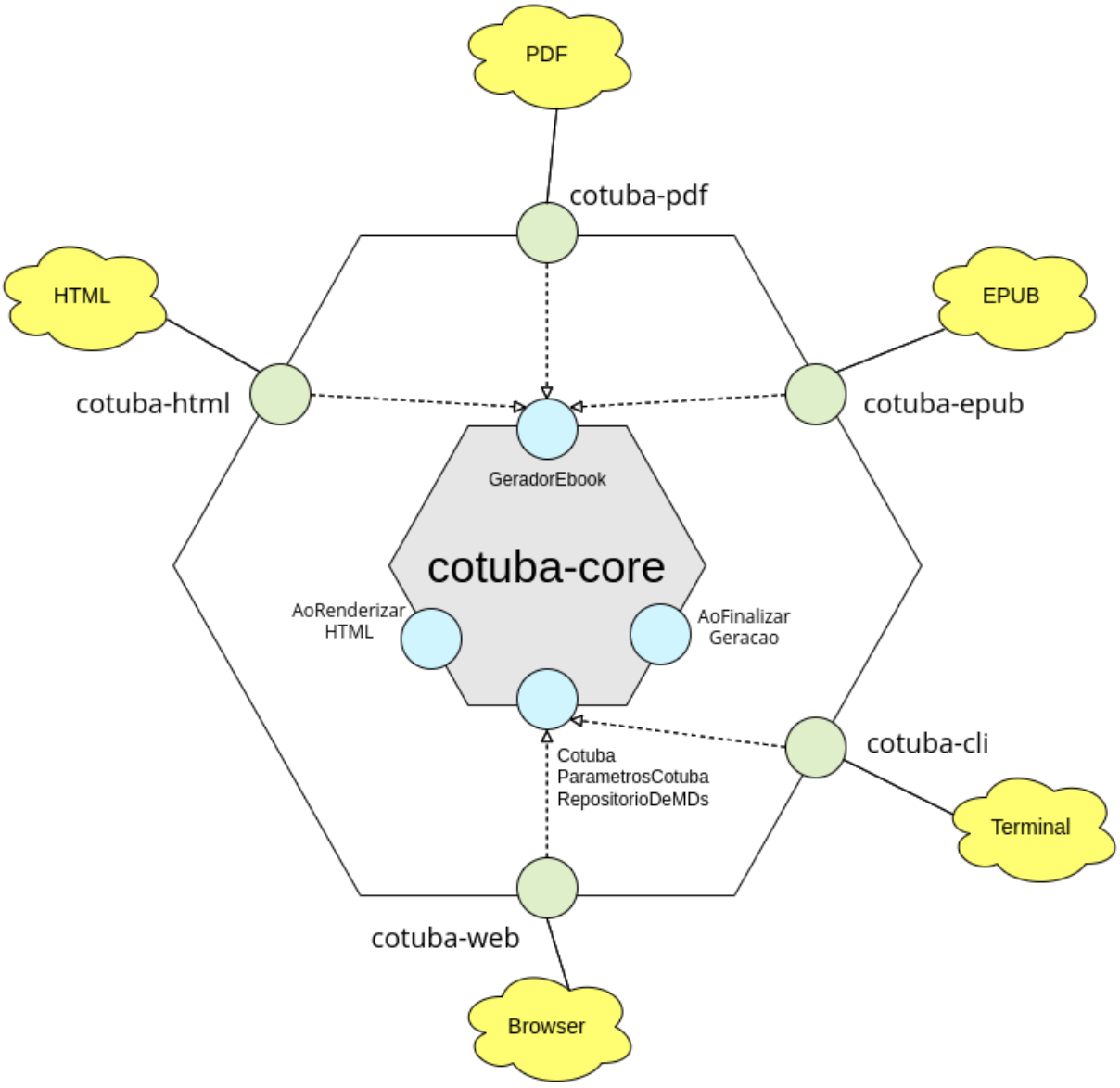

Figura 13.2: Cotuba como um hexágono

Poderíamos definir mais módulos, encaixando-os nos conectores já existentes. Por exemplo, poderíamos definir um módulo `cotuba-azw`, que gera um arquivo `.azw`, adequado para ser lido no leitor de ebooks Amazon Kindle. Para isso, o módulo `cotuba-core` não precisaria ser modificado.

Os conectores de plugins do Cotuba

O módulo `cotuba-core` define também conectores para plugins:

- `AoRenderizarHTML`, um conector que permite ações logo após a renderização dos arquivos Markdown em HTML, usado pelo módulo `tema-paradizo`.
- `AoFinalizarGeracao`, um conector que permite ações após a geração do ebook, usado pelo módulo `estatisticas-ebook`.

Os módulos `tema-paradizo` e `estatisticas-ebook` seriam fornecidos por times de desenvolvedores diferentes dos que desenvolvem o Cotuba.

13.3 Arquitetura de Plugins

Sob uma outra perspectiva, os *Ports and Adapters* estão relacionados a uma **arquitetura de plugins**.

Uma interface de linha comando seria um plugin. Uma aplicação Web seria outro plugin. A persistência em um banco de dados relacional, outro plugin. Uma possível persistência em um *datastore* NoSQL, mais um plugin. Integrações com outros sistemas, mais plugins.

Usando interfaces, invertemos as dependências para que todos dependam do código de negócio que, por sua vez, não depende de ninguém. As dependências sempre apontam na direção das regras de negócio, nunca ao contrário. Cada mecanismo de entrega seria um plugin diferente.

No livro *Clean Architecture* (MARTIN, 2017), Uncle Bob argumenta em favor de arquiteturas de plugins. O autor diz que a história das tecnologias de desenvolvimento de software é sobre como criar plugins de maneira conveniente para estabelecer uma arquitetura de sistemas escalável e manutenível. Para Uncle Bob, organizar o código da UI como um plugin torna possível plugar muitos tipos diferentes de UIs baseadas na Web, em cliente/servidor, em SOA, em consoles ou em qualquer outro tipo de UI. O mesmo vale para o Banco de Dados que, se for tratado como um plugin, pode ser trocado por qualquer um dos vários Bancos de Dados relacionais, NoSQL ou baseados em sistemas de arquivos ou qualquer outro tipo que julgarmos necessário no futuro.

Entretanto, Uncle Bob afirma que essas trocas de tecnologias podem não ser triviais. Se o deploy inicial de nosso sistema for baseado na Web, escrever o plugin para uma interface cliente/servidor pode ser desafiador. É provável que algumas das

comunicações entre as regras de negócio e a nova UI tenham que ser refeitas. Mesmo assim, para Uncle Bob, ao organizarmos o código inicial em uma estrutura de plugins, fazemos com que essa mudança de tecnologias seja prática.

Mark Richards e Neal Ford, em seu livro *Fundamentals of Software Architecture* (RICHARDS; FORD, 2020), chamam esse estilo arquitetural de *Microkernel Architecture* e o indicam para aplicações que são empacotadas, baixadas e instaladas em um deploy monolítico, comumente na máquina do usuário ou de empresas terceiras.

Em uma arquitetura de Microkernel, a aplicação é dividida entre um núcleo com funcionalidades mínimas e plugins, que implementam processamento especializado e funcionalidades adicionais. São dados alguns exemplos de aplicações que seguem esse estilo: a IDE Eclipse IDE; o servidor de integração contínua Jenkins; a ferramenta de análise estática de código PMD; o servidor de aplicação Wildfly; a ferramenta de gerenciamento de projetos Jira, que pode ser instalada nos *data centers* de empresas clientes; além de navegadores como o Chrome e o Firefox.

Richard e Fords avaliam que uma arquitetura de Microkernel tem custo e simplicidade como principais forças. Escalabilidade, tolerância a falhas e elasticidade seriam as principais fraquezas dessa arquitetura. Os autores a avaliam como acima da média nos quesitos testabilidade, implantabilidade, confiabilidade, performance, modularidade e capacidade de evolução.

13.4 Barreiras arquiteturais

No artigo *What is Software Architecture?* (MALAN, 2017), Ruth Malan associa Arquitetura Hexagonal à arquitetura de fortes medievais. Em um forte medieval, são criadas várias **barreiras arquiteturais**, como fossos e muralhas, para proteger a parte interna da fortificação. Muitas vezes, essas barreiras são concêntricas, criando várias camadas de proteção. Para permitir o acesso à parte interna, as muralhas têm portões e os fossos têm pontes.

Figura 13.3: Forte dos Reis Magos, Natal/RN. Fonte: Iphan

Para proteger a lógica de negócio de dependências externas e voláteis, em uma Arquitetura Hexagonal, criamos barreiras

arquiteturais por meio de abstrações (os conectores). As pontes para as dependências externas são feitas através dos adaptadores. Os conectores e adaptadores criam uma barreira arquitetural que permite pontos de acesso plugáveis.

> *"Organizar nossos sistemas em uma arquitetura de plugins cria firewalls através dos quais mudanças não podem ser propagadas. Se a UI é plugada nas regras de negócio, então mudanças na UI não afetam o negócio.*
>
> *Barreiras (boundaries, ou limites, no original) são traçadas onde há um eixo de mudança. Os componentes de um lado da barreira mudam a taxas diferentes e por motivos diferentes dos componentes do outro lado da barreira.*
>
> *UIs mudam em tempos e taxas diferentes das regras de negócio, então deve haver uma barreira entre eles. Regras de negócio mudam em tempos e taxas diferentes de frameworks de injeção de dependência, então deve haver uma barreira entre eles."*
>
> Uncle Bob, no livro *Clean Architecture* (MARTIN, 2017)

13.5 Clean Architecture

A arquitetura descrita por Uncle Bob no livro *Clean Architecture* (MARTIN, 2017) é bastante semelhante à hexagonal. O objetivo da *Clean Architecture* é definir uma abordagem arquitetural que torna a aplicação:

- **independente de frameworks**: um framework não é sua aplicação. A estrutura de diretórios e as restrições do design do nosso código não deveriam ser determinadas por um framework. Frameworks deveriam ser usados apenas como ferramentas para que a aplicação cumpra suas necessidades.
- **independente da UI**: a UI muda mais frequentemente que o resto do sistema. Além disso, é possível termos diferentes UIs para as mesmas regras de negócio.
- **independente de BD**: as regras de negócio não devem depender de um Banco de Dados específico. Devemos possibilitar a troca, de maneira fácil, de Oracle ou SQL Server para MongoDB, CouchDB, Neo4J ou qualquer outro BD.
- **testável**: deve ser possível testar as regras de negócio diretamente, sem a necessidade de usar uma UI, BD ou servidor Web.

Para isso, Uncle Bob define visualmente alguns círculos concêntricos:

- *Entities*: o círculo mais interno, contém regras de negócio críticas, que dizem respeito a diversas aplicações dentro da mesma empresa.
- *Use Cases*: usa as *Entities* para prover regras de negócio para uma aplicação específica. Casos de uso ou histórias de usuário são representados como objetos.

- *Interface Adapters*: usam os *Use Cases* e converte dados de/para dispositivos específicos.
- *Frameworks and Drivers*: o círculo mais externo, contém código específico de BDs, UI, sistemas externos e outros dispositivos.

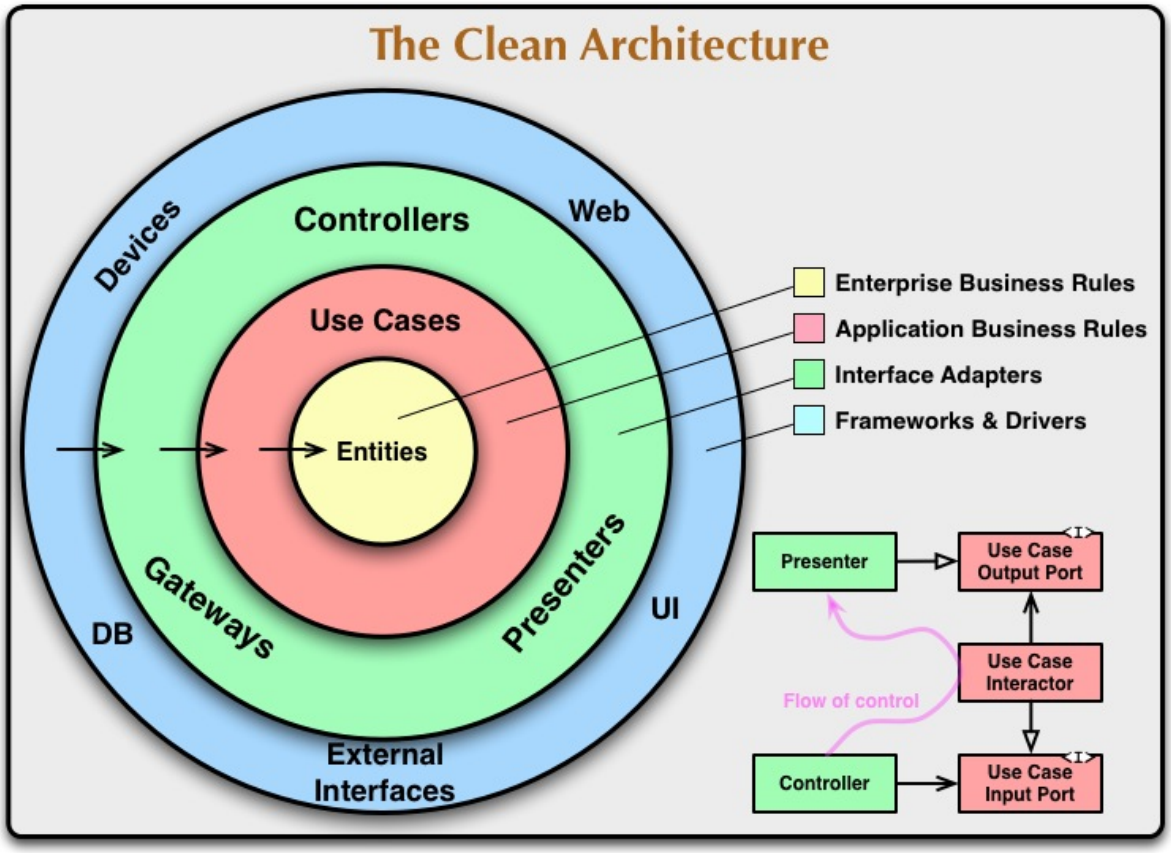

Figura 13.4: A Arquitetura Limpa

A Regra da Dependência

Conforme mencionamos no capítulo de DIP, Uncle Bob define o que chama de "regra da dependência": dependências devem apontar apenas para dentro, em direção às regras de negócio.

Um círculo interno não deve conhecer classes, métodos ou formatos de dados de círculos externos. Nenhuma mudança em um círculo externo deve afetar os círculos internos. Para permitir isso, um círculo interno deve inverter as dependências, fornecendo

abstrações que são implementadas pelo código de círculos externos.

Uncle Bob explica que: *"Para desenhar barreiras em uma arquitetura de software, primeiro você particiona o sistema em componentes. Alguns desses componentes são regras de negócio centrais; outros são plugins que contêm funções necessárias que não estão diretamente ligadas ao negócio. Então, você organiza o código desses componentes de maneira que as setas entre eles apontem em uma direção: em direção às regras de negócio."* (MARTIN, 2017)

Outras arquiteturas semelhantes

Ainda no livro *Clean Architecture* (MARTIN, 2017), Uncle Bob cita algumas outras arquiteturas semelhantes à Clean Architecture (e à Hexagonal):

- DCI (*Data, Context and Interaction*), descrita por James Coplien, pioneiro dos Design Patterns, e Trygve Reenskaug, criador do MVC.
- BCE (*Boundary-Control-Entity*), introduzida por Ivar Jacobson, um dos criadores do UML.

Para saber mais: Microsserviços e o Monólito Modular

O uso de **microsserviços** é uma abordagem de decomposição do código de um sistema em que as barreiras arquiteturais são serviços independentes e autônomos. Diferentes dos módulos, cuja comunicação é feita por chamadas de métodos no mesmo processo, os serviços têm uma comunicação interprocesso, pela rede. Entre as vantagens estão a possibilidade de implantação independente, a opção por tecnologias heterogêneas, a maior tolerância a falhas e mais facilidade ao escalar. As grandes desvantagens, porém, são a complexidade da solução e um possível impacto na performance pela comunicação em rede.

O "antagonista" dos microsserviços é o **monólito**: um único executável é implantado em produção. Para publicar uma nova versão, o sistema todo tem que ser reiniciado. Uma falha na versão publicada afeta todo o sistema. Há o perigo das dependências entre as fatias do código saírem do controle. As grandes vantagens são a simplicidade e facilidade de evolução e refatoração do código.

Em seu artigo *Microservices and Jars* (MARTIN, 2014b), Uncle Bob escreve: *"Não pule para microsserviços só porque parece legal. Antes, segregue o sistema em JARs usando uma arquitetura de plugins. Se isso não for suficiente, considere a introdução de limites de serviço (service boundaries) em pontos estratégicos."*

No livro *Building Microservices* (NEWMAN, 2015), Sam Newman argumenta que o uso de módulos é uma boa maneira de começar uma base de código, porque eles ajudam a manter junto código relacionado e a reduzir o acoplamento com outros módulos. Os módulos poderiam ser modelados a partir de *bounded contexts* - termo do *Domain Driven Design* para os diferentes contextos de negócio de um sistema. Cada módulo passa a ser um candidato a um microsserviço. Para Newman, *"obter os limites incorretos entre os serviços pode ser caro, então é importante esperar que as coisas se estabilizem à medida que você se familiariza com um novo domínio."*

Martin Fowler, em seu artigo *Monolith First* (FOWLER, 2015), descreve uma abordagem em que o desenvolvimento é iniciado em um monólito para só tardiamente o quebrar em microsserviços. Com o projeto em andamento, as necessidades de negócio e, consequentemente, as barreiras arquiteturais ficam mais claras. Fowler diz que *"a maneira lógica é ter um design cuidadoso do monólito, prestando atenção à modularidade do software, tanto no nível de APIs quanto no modo como os dados são armazenados. Faça isso bem e será relativamente simples migrar para microsserviços"*. Porém, Fowler indica que a abordagem de um monólito modular não é tão corriqueira. Descartar completamente o

monólito ou retirar apenas alguns serviços de maior granularidade são abordagens mais comuns.

13.6 Contraponto

No episódio *Clean Architecture #254* (ARCOVERDE et al., 2021) do podcast Hipsters Ponto Tech, Roberta Arcoverde faz uma provocação interessante ao mencionar que Arquitetura Hexagonal é termo ruim porque nem sempre há um hexágono formado por seis componentes e que trata mais de design (onde fica o código) do que de arquitetura de software (quais são os componentes e como eles interagem). Maurício Linhares aponta que, em um ambiente de sistemas distribuídos sendo executados na nuvem, a arquitetura de sistemas passa a ser uma preocupação crucial. Ele complementa dizendo que barreiras entre o código de domínio e o que não está relacionado é importante, mas é preciso encontrar um meio termo, já que muita coisa acaba vazando de interfaces que aparentemente escondem detalhes. Um exemplo dado por Linhares é a migração de um Bancos de Dados Relacional para um Banco de Dados NoSQL baseado em Documentos que causa muita dor de cabeça em quem já tentou fazer: os jeitos de interagir, as respostas e o comportamento dos sistemas são completamente diferentes. Para Linhares, a suposta flexibilidade, na prática, não ajuda tanto assim.

No vídeo *Clean architecture: Provavelmente você não quer isso* (SOUZA, 2021), Alberto Souza argumenta que não faz sentido, na grande maioria dos casos, o empenho em deixar uma aplicação corporativa completamente desacoplada de tecnologias e preparada para a mudança de detalhes técnicos sem impacto no código de domínio. Em especial, em microsserviços, que são aplicações pequenas e com o valor de negócio bem delimitado. No vídeo, Alberto mostra a complexidade do código do Spring que, por ser um framework bastante flexível, precisa ser desacoplado de tecnologias específicas. Para Alberto, adotar um estilo arquitetural inspirado na Clean Architecture leva a produzir muito mais código do que o necessário. Além disso, ele argumenta que a troca de tecnologias é

algo bastante raro em uma aplicação corporativa e não deve ser a prioridade da arquitetura adotada. Para Alberto, a questão crucial é: o esforço da troca de tecnologias é consideravelmente maior que todo o esforço acumulado para proteger o código da mudança de tecnologia?

13.7 Conclusão

Começamos o livro com uma pergunta: para que serve Orientação a Objetos? Há duas respostas para essa pergunta.

Uma das vertentes está preocupada em alinhar a linguagem de negócio com o que é representado no código, usando Orientação a Objetos para criar modelos de domínio e delimitando esses modelos em contextos para que não haja inconsistências. É o caso do *Domain-Driven Design* (DDD).

Uma outra vertente foca nos aspectos de gerenciamento de dependências. Ao aplicarmos os princípios SOLID junto aos princípios de acoplamento e coesão de módulos de Uncle Bob, criamos barreiras arquiteturais que desacoplam as regras de negócios de detalhes técnicos. No livro *Clean Architecture* (MARTIN, 2017), Uncle Bob afirma: *"OO é a habilidade, por meio do uso de polimorfismo, de ganhar controle absoluto de todas as dependências de código de um sistema. Isso permite que a pessoa responsável pela arquitetura crie uma arquitetura de plugins, em que módulos que contêm diretrizes (policies, no original) de alto nível sejam independentes de módulos que contêm detalhes de baixo nível. Os detalhes de baixo nível são relegados a módulos de plugin que podem ser implantados e desenvolvidos independentemente dos módulos que contêm diretrizes de alto nível"*.

Essas duas vertentes são complementares: podemos usar ideias do DDD para modelar as regras de negócio no núcleo da aplicação e os princípios SOLID e de módulos para estruturar o código de maneira que esse núcleo fique isolado de detalhes técnicos.

E qual é o objetivo final? Uma arquitetura que permita código flexível, fácil de manter e que se adapta às mudanças de negócios. Mas é importante ressaltar que uma priorização exagerada da

flexibilidade pode levar a um aumento da complexidade e a uma dificuldade no entendimento do código.

Cabe a você decidir em quais pontos do sistema é preciso abraçar a mudança!

CAPÍTULO 14

Apêndice: O código inicial do Cotuba

Abaixo está o nosso ponto de partida neste livro, em que toda a lógica do Cotuba é feita em apenas uma classe, a `Main`. Este código pode ser encontrado em: https://github.com/alexandreaquiles/solid-na-pratica

Figura 14.1: Código inicial do Cotuba

```
package cotuba;

import java.io.File;
import java.io.IOException;
import java.nio.file.FileSystems;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.PathMatcher;
import java.nio.file.Paths;
import java.util.Comparator;
import java.util.List;
```

```java
import java.util.stream.Stream;

import org.apache.commons.cli.CommandLine;
import org.apache.commons.cli.CommandLineParser;
import org.apache.commons.cli.DefaultParser;
import org.apache.commons.cli.HelpFormatter;
import org.apache.commons.cli.Option;
import org.apache.commons.cli.Options;
import org.apache.commons.cli.ParseException;
import org.commonmark.node.AbstractVisitor;
import org.commonmark.node.Heading;
import org.commonmark.node.Node;
import org.commonmark.node.Text;
import org.commonmark.parser.Parser;
import org.commonmark.renderer.html.HtmlRenderer;

import com.itextpdf.html2pdf.HtmlConverter;
import com.itextpdf.kernel.pdf.PdfDocument;
import com.itextpdf.kernel.pdf.PdfWriter;
import com.itextpdf.layout.Document;
import com.itextpdf.layout.element.AreaBreak;
import com.itextpdf.layout.element.IBlockElement;
import com.itextpdf.layout.element.IElement;
import com.itextpdf.layout.property.AreaBreakType;

import nl.siegmann.epublib.domain.Book;
import nl.siegmann.epublib.domain.Resource;
import nl.siegmann.epublib.epub.EpubWriter;
import nl.siegmann.epublib.service.MediatypeService;

public class Main {

  public static void main(String[] args) {
    var options = new Options();

    var opcaoDeDiretorioDosMD = new Option("d", "dir", true,
        "Diretório que contém os arquivos md."
            + "Default: diretório atual.");
    options.addOption(opcaoDeDiretorioDosMD);

    var opcaoDeFormatoDoEbook = new Option("f", "format", true,
```

```
    "Formato de saída do ebook. Pode ser: pdf ou epub."
        + " Default: pdf");
options.addOption(opcaoDeFormatoDoEbook);

var opcaoDeArquivoDeSaida = new Option("o", "output", true,
    "Arquivo de saída do ebook. Default: book.{formato}.");
options.addOption(opcaoDeArquivoDeSaida);

var opcaoModoVerboso = new Option("v", "verbose", false,
    "Habilita modo verboso.");
options.addOption(opcaoModoVerboso);

CommandLineParser cmdParser = new DefaultParser();
var ajuda = new HelpFormatter();
CommandLine cmd;

try {
  cmd = cmdParser.parse(options, args);
} catch (ParseException e) {
  System.err.println(e.getMessage());
  ajuda.printHelp("cotuba", options);
  System.exit(1);
  return;
}

Path diretorioDosMD;
String formato;
Path arquivoDeSaida;
boolean modoVerboso = false;

try {

  String nomeDoDiretorioDosMD = cmd.getOptionValue("dir");

  if (nomeDoDiretorioDosMD != null) {
    diretorioDosMD = Paths.get(nomeDoDiretorioDosMD);
    if (!Files.isDirectory(diretorioDosMD)) {
      throw new IllegalArgumentException(nomeDoDiretorioDosMD
        + " não é um diretório.");
    }
  } else {
```

```
  Path diretorioAtual = Paths.get("");
  diretorioDosMD = diretorioAtual;
}

String nomeDoFormatoDoEbook = cmd.getOptionValue("format");

if (nomeDoFormatoDoEbook != null) {
  formato = nomeDoFormatoDoEbook.toLowerCase();
} else {
  formato = "pdf";
}

String nomeDoArquivoDeSaidaDoEbook =
  cmd.getOptionValue("output");
if (nomeDoArquivoDeSaidaDoEbook != null) {
  arquivoDeSaida = Paths.get(nomeDoArquivoDeSaidaDoEbook);
} else {
  arquivoDeSaida = Paths.get("book." +
      formato.toLowerCase());
}
if (Files.isDirectory(arquivoDeSaida)) {
  // deleta arquivos do diretório recursivamente
  Files.walk(arquivoDeSaida)
      .sorted(Comparator.reverseOrder())
      .map(Path::toFile).forEach(File::delete);
} else {
  Files.deleteIfExists(arquivoDeSaida);
}

modoVerboso = cmd.hasOption("verbose");

if ("pdf".equals(formato)) {
  try(var writer = new PdfWriter(Files
          .newOutputStream(arquivoDeSaida));
    var pdf = new PdfDocument(writer);
    var pdfDocument = new Document(pdf)) {

    PathMatcher matcher = FileSystems.getDefault()
        .getPathMatcher("glob:**/*.md");
    try (Stream<Path> arquivosMD =
            Files.list(diretorioDosMD)) {
```

```
arquivosMD
  .filter(matcher::matches)
  .sorted()
  .forEach(arquivoMD -> {
    Parser parser = Parser.builder().build();
    Node document = null;
    try {
      document = parser.parseReader(
            Files.newBufferedReader(arquivoMD));
      document.accept(new AbstractVisitor() {
        @Override
        public void visit(Heading heading) {
          if (heading.getLevel() == 1) {
            // capítulo
            String tituloDoCapitulo = ((Text) heading
                .getFirstChild()).getLiteral();
            // TODO: usar título do capítulo
          } else if (heading.getLevel() == 2) {
            // seção
          } else if (heading.getLevel() == 3) {
            // título
          }
        }

      });
    } catch (Exception ex) {
      throw new IllegalStateException(
        "Erro ao fazer parse do arquivo "
            + arquivoMD, ex);
    }

    try {
      HtmlRenderer renderer = HtmlRenderer.builder()
          .build();
      String html = renderer.render(document);

      List<IElement> convertToElements =
          HtmlConverter.convertToElements(html);
      for (IElement element : convertToElements) {
        pdfDocument.add((IBlockElement) element);
      }
```

```
            // TODO: não adicionar página depois do último capítulo
            pdfDocument.add(
                new AreaBreak(AreaBreakType.NEXT_PAGE));

          } catch (Exception ex) {
            throw new IllegalStateException(
              "Erro ao renderizar para HTML o arquivo "
                  + arquivoMD, ex);
          }

        });
    } catch (IOException ex) {
      throw new IllegalStateException(
        "Erro tentando encontrar arquivos .md em "
            + diretorioDosMD.toAbsolutePath(), ex);
    }

  } catch (Exception ex) {
    throw new IllegalStateException(
      "Erro ao criar arquivo PDF: "
          + arquivoDeSaida.toAbsolutePath(), ex);
  }

} else if ("epub".equals(formato)) {
  var epub = new Book();

  PathMatcher matcher = FileSystems.getDefault()
      .getPathMatcher("glob:**/*.md");
  try (Stream<Path> arquivosMD =
          Files.list(diretorioDosMD)) {
    arquivosMD
      .filter(matcher::matches)
      .sorted()
      .forEach(arquivoMD -> {
        Parser parser = Parser.builder().build();
        Node document = null;
        try {
          document = parser.parseReader(
                Files.newBufferedReader(arquivoMD));
          document.accept(new AbstractVisitor() {
            @Override
```

```
            public void visit(Heading heading) {
              if (heading.getLevel() == 1) {
                // capítulo
                String tituloDoCapitulo = ((Text) heading
                    .getFirstChild()).getLiteral();
                // TODO: usar título do capítulo
              } else if (heading.getLevel() == 2) {
                // seção
              } else if (heading.getLevel() == 3) {
                // título
              }
            }

          });
        } catch (Exception ex) {
          throw new IllegalStateException(
            "Erro ao fazer parse do arquivo "
                + arquivoMD, ex);
        }

        try {
          HtmlRenderer renderer = HtmlRenderer.builder()
              .build();
          String html = renderer.render(document);

          // TODO: usar título do capítulo
          epub.addSection("Capítulo", new Resource(
                html.getBytes(), MediatypeService.XHTML));

        } catch (Exception ex) {
          throw new IllegalStateException(
            "Erro ao renderizar para HTML o arquivo "
                + arquivoMD, ex);
        }
      });
    } catch (IOException ex) {
      throw new IllegalStateException(
        "Erro tentando encontrar arquivos .md em "
            + diretorioDosMD.toAbsolutePath(), ex);
    }
```

```
        var epubWriter = new EpubWriter();

        try {
          epubWriter.write(epub,
              Files.newOutputStream(arquivoDeSaida));
        } catch (IOException ex) {
          throw new IllegalStateException(
            "Erro ao criar arquivo EPUB: "
                + arquivoDeSaida.toAbsolutePath(), ex);
        }
      } else {
        throw new IllegalArgumentException(
          "Formato do ebook inválido: " + formato);
      }

      System.out.println("Arquivo gerado com sucesso: " +
          arquivoDeSaida);

    } catch (Exception ex) {
      System.err.println(ex.getMessage());
      if (modoVerboso) {
        ex.printStackTrace();
      }
      System.exit(1);
    }
  }

}
```

CAPÍTULO 15

Apêndice: Referências

ABELSON, Harold; SUSSMAN, Gerald Jay; SUSSMAN, Julie. **Structure and Interpretation of Computer Programs**. 2. ed. Cambridge, MA. MIT Press, 1996.

ANICHE, Maurício. **Orientação a Objetos e SOLID para Ninjas: Projetando classes flexíveis**. São Paulo. Casa do Código, 2015.

ARCOVERDE, Roberta et al. **Clean Architecture - Hipsters Ponto Tech #254**. 2021. https://hipsters.tech/clean-architeture-hipsters-ponto-tech-254. Acesso em: 06 abr. 2022.

BECK, Kent. **Extreme Programming Explained: Embrace Change**. Boston, MA. Addison-Wesley Professional, 2000.

BECK, Kent. **Test Driven Development: By Example**. Boston, MA. Addison-Wesley Professional, 2002.

BLOCH, Joshua. **Effective Java: Programming Language Guide**. Boston, MA. Addison-Wesley Professional, 2001.

BROWN, Simon. **Modularity and testability: Designing software requires conscious effort; let's not stop thinking**. 2014. http://www.codingthearchitecture.com/2014/10/01/modularity_and_testability.html. Acesso em: 06 abr. 2022.

CHIDAMBER, Shyam; KEMERER, Chris. **A metrics suite for object oriented design**. IEEE Software, Jun. 1994. https://sites.pitt.edu/~ckemerer/CK%20research%20papers/MetricForOOD_ChidamberKemerer94.pdf. Acesso em: 06 abr. 2022.

CIRILLO, Francesco. **Anti-IF Campaign**. 2009. https://web.archive.org/web/20090401163659/http://www.antiifcampaign.com/. Acesso em: 06 abr. 2022.

COCKBURN, Alistair. **Hexagonal Architecture**. 2005. https://web.archive.org/web/20090122225311/http://alistair.cockburn.us/Hexagonal+architecture. Acesso em: 06 abr. 2022.

COPELAND, David B. **SOLID is not solid**. 2019. https://solid-is-not-solid.com/. Acesso em: 06 abr. 2022.

FEATHERS, Michael. **Negative Architecture**. 2018. https://michaelfeathers.silvrback.com/negative-architecture. Acesso em: 06 abr. 2022.

RICHARDS, Mark; FORD, Neal. **Fundamentals of Software Architecture: an Engineering Approach**. Sebastopol, CA. O'Reilly Media, 2020.

FOWLER, Martin et al. **Refactoring: Improving the Design of Existing Code**. Boston, MA. Addison-Wesley Professional, 1999.

FOWLER, Martin et al. **Patterns of Enterprise Application Architecture**. Boston, MA. Addison-Wesley Professional, 2002.

FOWLER, Martin. **SoftwareDevelopmentAttitude**. 2004. https://www.martinfowler.com/bliki/SoftwareDevelopmentAttitude.html. Acesso em: 06 abr. 2022.

FOWLER, Martin. **MonolithFirst**. 2015. https://martinfowler.com/bliki/MonolithFirst.html. Acesso em: 06 abr. 2022.

FREEMAN, Steve; PRICE, Nat. **Growing Object-Oriented Software, Guided by Tests**. Boston, MA. Addison-Wesley Professional, 2009.

GAMMA, Erich et al. **Design Patterns: Elements of Reusable Object-Oriented Software**. Boston, MA. Addison-Wesley Professional, 1994.

HENNEY, Kevlin. **YOW! 2013 Kevlin Henney - The SOLID Design Principles Deconstructed**. 2013. https://www.youtube.com/watch?v=tMW08JkFrBA. Acesso em: 06 abr. 2022.

HUNT, Andrew; THOMAS, David. **The Pragmatic Programmer**: from Journeyman to Master. Boston, MA. Addison-Wesley Professional, 1999.

HUNT, Andrew; THOMAS, David. **The Art of Enbugging**. IEEE Software Construction, Jan/Fev. 2003. https://media.pragprog.com/articles/jan_03_enbug.pdf. Acesso em: 06 abr. 2022.

KAMINSKI, Ted. **Deconstructing SOLID design principles**. 2019. https://web.archive.org/web/20200621040933/https://www.tedinski.com/2019/04/02/solid-critique.html. Acesso em: 06 abr. 2022.

KNOERNSCHILD, Kirk. **Java Application Architecture: Modularity Patterns with examples using OSGi**. Upper Saddle River, NJ. Prentice Hall, 2012.

LARMAN, Craig. **Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development**. 3. ed. Upper Saddle River, NJ. Prentice Hall, 2004.

LARMAN, Craig. **Protected Variation**._ IEEE Software, Mai./Jun. 2001. https://www.martinfowler.com/ieeeSoftware/protectedVariation.pdf. Acesso em: 06 abr. 2022.

LISKOV, Barbara. **Data Abstraction and Hierarchy**. 1988. https://www.cs.tufts.edu/~nr/cs257/archive/barbara-liskov/data-abstraction-and-hierarchy.pdf. Acesso em: 06 abr. 2022.

MAK, Sander; BAKKER, Paul. **Java 9 Modularity**. Sebastopol, CA. O'Reilly Media, 2017.

MALAN, Ruth. **What is Software Architecture?** 2017. https://web.archive.org/web/20190525014713/http://www.bredemeyer.com/whatis.htm. Acesso em: 06 abr. 2022.

MARTIN, Robert Cecil. **The Ten Commandments of OO Programming**. 1995. https://groups.google.com/g/comp.object/c/WICPDcXAMG8/m/EbGa2Vt-7q0J. Acesso em: 06 abr. 2022.

MARTIN, Robert Cecil. **The Open-Closed Principle**. The C++ Report, Jan. 1996a. https://drive.google.com/file/d/0BwhCYaYDn8EgN2M5MTkwM2EtNWFkZC00ZTI3LWFjZTUtNTFhZGZiYmUzODc1/view. Acesso em: 06 abr. 2022.

MARTIN, Robert Cecil. **The Liskov Substitution Principle**. The C++ Report, Mar. 1996b. https://drive.google.com/file/d/0BwhCYaYDn8EgNzAzZjA5ZmItNjU3NS00MzQ5LTkwYjMtMDJhNDU5ZTM0MTlh/view. Acesso em: 06 abr. 2022.

MARTIN, Robert Cecil. **The Dependency Inversion Principle**. The C++ Report, Jun. 1996c.

https://drive.google.com/file/d/0BwhCYaYDn8EgMjdlMWIzNGUtZTQ0NC00ZjQ5LTkwYzQtZjRhMDRlNTQ3ZGMz/view. Acesso em: 06 abr. 2022.

MARTIN, Robert Cecil. **The Interface Segregation Principle**. The C++ Report, Aug. 1996d. https://drive.google.com/file/d/0BwhCYaYDn8EgOTViYjJhYzMtMzYxMC00MzFjLWJjMzYtOGJiMDc5N2JkYmJi/view. Acesso em: 06 abr. 2022.

MARTIN, Robert Cecil. **Granularity**. The C++ Report, Nov. 1996e. https://drive.google.com/file/d/0BwhCYaYDn8EgOGM2ZGFhNmYtNmE4ZS00OGY5LWFkZTYtMjE0ZGNjODQ0MjEx/view. Acesso em: 06 abr. 2022.

MARTIN, Robert Cecil. **Stability**. The C++ Report, Jan. 1997. https://drive.google.com/file/d/0BwhCYaYDn8EgZjI3OTU4ZTAtYmM4Mi00MWMyLTgxN2YtMzk5YTY1NTViNTBh/view. Acesso em: 06 abr. 2022.

MARTIN, Robert Cecil. **Design Principles and Design Patterns**. 2000. https://drive.google.com/file/d/1QkpGscUCg2M8paF_90J-XN-H9sZ0A3PG/view. Acesso em: 06 abr. 2022.

MARTIN, Robert Cecil. **UML for Java Programmers**. Upper Saddle River, NJ. Prentice Hall, 2003.

MARTIN, Robert Cecil. **The Principles of OOD**. 2005. https://web.archive.org/web/20060417231754/http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod. Acesso em: 06 abr. 2022.

MARTIN, Robert Cecil. **Agile Principles, Patterns, and Practices in C#**. Upper Saddle River, NJ. Prentice Hall, 2006.

MARTIN, Robert Cecil. **Getting a SOLID start**. 2009. https://sites.google.com/site/unclebobconsultingllc/getting-a-solid-start. Acesso em: 06 abr. 2022.

MARTIN, Robert Cecil. **The Single Responsibility Principle**. 2014a. https://blog.cleancoder.com/uncle-bob/2014/05/08/SingleReponsibilityPrinciple.html. Acesso em: 06 abr. 2022.

MARTIN, Robert Cecil. **Microservices and Jars**. 2014b. https://blog.cleancoder.com/uncle-bob/2014/09/19/MicroServicesAndJars.html. Acesso em: 06 abr. 2022.

MARTIN, Robert Cecil. **Clean Architecture: A Craftsman's Guide to Software Structure and Design**. Upper Saddle River, NJ. Prentice Hall, 2017.

MEYER, Bertrand. **Object-Oriented Software Construction**. Upper Saddle River, NJ. Prentice Hall, 1988.

MÜLLER, Philip. **Under Deconstruction: The State of Shopify's Monolith**. 2020. https://shopify.engineering/shopify-monolith. Acesso em: 06 abr. 2022.

NEWMAN, Sam. **Building Microservices: Designing Fine-Grained Systems**. Sebastopol, CA. O'Reilly Media, 2015.

NORTH, Dan. **CUPID — the back story**. 2021. https://dannorth.net/2021/03/16/cupid-the-back-story/. Acesso em: 06 abr. 2022.

PARLOG, Nicolai. **The Java Module System**. Greenwich, CT. Manning Publications, 2019.

PARNAS, David L. **On the Criteria To Be Used in Decomposing Systems into Modules**. Communications of the ACM, Dez. 1972. https://www.win.tue.nl/~wstomv/edu/2ip30/references/criteria_for_modularization.pdf. Acesso em: 06 abr. 2022.

RONEI, Gabriel. **Monte seu Type-Safe builder**. 2021. https://dev.to/gabrielronei/monte-seu-type-safe-builder-1i71. Acesso em: 06 abr. 2022.

SILVEIRA, Paulo. **Como não aprender orientação a objetos: Herança**. 2006. https://www.alura.com.br/artigos/como-nao-aprender-orientacao-a-objetos-heranca. Acesso em: 06 abr. 2022.

SILVEIRA, Guilherme. **Como não aprender orientação a objetos: o excesso de ifs**. 2011. https://www.alura.com.br/artigos/como-nao-aprender-orientacao-a-objetos-o-excesso-de-ifs. Acesso em: 06 abr. 2022.

SOUZA, Alberto. **SOLID #1: Uma reflexão sobre o Princípio da responsabilidade única**. 2020. https://www.youtube.com/watch?v=GGe0o_v5vjM. Acesso em: 06 abr. 2022.

SOUZA, Alberto. **Clean architecture: Provavelmente você não quer isso**. 2021. https://www.youtube.com/watch?v=SQfpiDlYd0g. Acesso em: 06 abr. 2022.

SVRTAN, Damir; MAKAGON, Sergii. **Ready for changes with Hexagonal Architecture**. 2020. https://netflixtechblog.com/ready-for-changes-with-hexagonal-architecture-b315ec967749. Acesso em: 06 abr. 2022.

VANDERBURG, Glenn. **Cohesion**. 2011. https://vanderburg.org/blog/2011/01/31/cohesion.html. Acesso em: 06 abr. 2022.

WALLS, Craig. **Modular Java: Creating Flexible Applications with OSGi and Spring**. Raleigh, NC. The Pragmatic Bookshelf, 2009.